



COMPUTERWORLD[®]

November 2015 | www.computerworld.com

KNOWLEDGE PACK



INTRODUCTION
to
Windows
PowerShell

Introduction to Windows PowerShell

THE POWERSHELL scripting language is a must-have for every Windows administrator nowadays. The most current version of Windows Server is accessible primarily via PowerShell and only secondarily via the server's GUI.

Plus, it can't hurt to learn a new way of automating some of the more manual-intensive processes you do, including adding or deleting users.

With this guide, you'll learn everything from how to get started to more advanced ways of using PowerShell. And, once you've mastered the basics, we provide plenty of additional resources to help you learn even more.

So, where to start? That depends on how you like to learn. If you're the type of person who wants to know a lot right away, then our most in-depth information is the "Introduction to Windows PowerShell" book excerpt by Brian Knittel that begins on page 3.

If, however, you want a quicker and more hands-on type of introduction, check out "PowerShell: The basics" on page 23; that should have you up and running in no time.

For more visual folks, check out "9 useful PowerShell tools" on page 39, or the scriptshow that shows how to perform five specific tasks on page 50. And anyone who already knows something about the language but wants to learn even more might check out our primer on how to use objects starting on page 44.

Whatever your style, we're hoping there's something in here that will suit and that will help you learn a new skill. Enjoy!

—Johanna Ambrosio, technologies editor

contents

- ➔ [Introduction to Windows PowerShell](#)
- ➔ [Getting started with PowerShell: The basics](#)
- ➔ [PowerShell for beginners: Scripts and loops](#)
- ➔ [9 useful PowerShell tools](#)
- ➔ [Understanding and using objects in PowerShell](#)
- ➔ [Scriptshow: Performing five common administrative tasks using Windows PowerShell](#)

Introduction to Windows PowerShell

Microsoft's Windows PowerShell is, well, powerful, but also a bit peculiar. Here's what you need to know to get started with this robust command and scripting environment. By Brian Knittel

This article is excerpted from the book [Windows 7 and Vista Guide to Scripting, Automation, and Command Line Tools](#), copyright Pearson Education, all rights reserved. Reprinted with permission.

IN 2006, Windows Script Host (WSH) and the Command Prompt shell got a new sibling when Microsoft released a completely new environment called Windows PowerShell. PowerShell has some similarities to the older tools: Like the Command Prompt shell, it lets you type commands interactively as well as put them into script/batch files, and you can easily use pipes and redirection to string files and programs together. Like WSH, it provides a powerful object-oriented programming language for complex scripting.

But PowerShell is a very different, strange animal, and it has powerful features that go way beyond what WSH and the Command Prompt offer. I show you what I mean in the following sections.

An Object-Oriented Command Shell

Here's one way that that PowerShell is different and strange. Recall that if you type `dir` into a regular Command Prompt window, the `dir` command spits out a bunch of text onto the screen, giving the names, sizes, and timestamps of the files in a directory. You can direct that text into a file using the `>` redirection operator, or you can “pipe” the text to another program using the `|` operator. For example, the command

```
dir /s/b | sort
```

lists the files in the current directory and its subdirectories as a stream of text, and the text is sent to the `sort` program that sorts them alphabetically. If you've used a command-line environment in nearly any computer operating system in the last, say, 30 years, this should be familiar.

Windows PowerShell command lines appear to work in the same way, but its commands deal with *objects* rather than just text — objects that represent files, folders,



Introduction to Windows PowerShell

Windows device drivers, network services, documents any of the hundreds of objects defined in the .NET Framework library). What flows from one command to the next, via that pipe symbol (`|`), is a stream of objects. The various PowerShell commands let you generate, manipulate, invoke methods on, change properties of, and extract information from these objects.

As an example, here's what happens when you type `dir` in a PowerShell window. The PowerShell `dir` command is actually an alias (shorthand) for the `Get-ChildItem` cmdlet. You can type either name and get the same result.

Wait, cmd-what? *Cmdlet*, pronounced "command-let," as in "cute tiny little command." PowerShell's built-in commands are called cmdlets. Maybe they were out of vowels up in Redmond that year? No, the reason for the odd name is that they're not completely built in to PowerShell, like `cmd.exe`'s commands, nor are they really completely independent of it, like `exe` files. They're implemented in a new way, so they needed a new word. In any case. . .

With no other arguments, `Get-ChildItem` emits File and Folder objects for all the files and subfolders in the current directory. If I type the command `dir` with no arguments, `Get-ChildItem` spits out a list of File and Folder objects, and because there is no pipe or output redirection, the results spill into the Windows PowerShell command window.

When objects land in the PowerShell window, PowerShell prints one line for each object, in a nice columnar format, listing the most important properties of each object. For File and Folder objects, this includes the Name, Length, LastWriteTime, and Mode properties. The resulting list looks like this:

```
Directory: C:\users\bknittel<br> Mode LastWriteTime Length Name<br>-----<br>-----<br> d-r-- 6/2/2010 1:27 PM Contacts<br> d-r-- 6/15/2010 11:35 PM Desktop<br> d-r-- 6/14/2010 8:47 AM Documents<br> d-r-- 6/2/2010 1:27 PM Downloads<br> :<br> d-r-- 6/2/2010 1:27 PM Videos<br> d-r-- 6/23/2010 11:01 AM Virtual Machines<br> -a--- 8/21/2009 6:07 PM 1889 listprops.vbs<br> -a--- 8/20/2009 12:34 PM 1020 nets.vbs<br> -a--- 8/20/2009 11:54 AM 3668 tempfile.txt<br> -a--- 9/14/2009 7:13 PM 1605 test.vbs<br> -a--- 6/23/2010 2:19 PM 3650 x
```

The headings at the top of the columns are just the names of the properties that are being displayed. The listing would be similar for any other type of object a cmdlet generated.

The thing to ponder here is that the `dir` command in the regular Command Prompt environment generates text in a fixed format. The `dir` cmdlet in Windows PowerShell generates a list of File and Folder objects, which PowerShell formats into a text listing after the fact.

You can redirect the output of a PowerShell cmdlet to a file, using the familiar `>` character, and the same thing occurs: the stream of objects gets formatted into a nice, text listing.

The innovation in PowerShell is what happens when you use the pipe redirection symbol (`|`). PowerShell lets you direct a stream of objects from one cmdlet to another, and they can change the properties of and call methods on these objects, doing real



Introduction to Windows PowerShell

work as they get passed along. It's only when the objects finally hit the screen that text appears. One cmdlet might generate a list of objects representing files, computers, services, and network objects; the next in the pipeline might filter and pass on just the ones that are of interest; and the next might call methods to perform actions on the objects. This is a truly unique feature of Windows PowerShell.

So, to do what used to require a loop in WSH:

```
fso = CreateObject("Scripting.FileSystemObject")<br> for each file in fso.GetFolder("c:\temp").Files<br>  file.Delete<br> next
```

you can do with a single command line in Windows PowerShell:

```
dir c:\temp | remove-item
```

`dir` generates a list of File and Folder objects representing the contents of directory `c:\temp` and passes it to the `remove-item` command, which deletes the “real world” thing behind any object it's passed.

You could also type the following:

```
(dir c:\temp).delete()
```

which generates the same stream of File and Folder objects and calls the `delete` method on each object. The result is the same: The files are deleted.

(And, as you might expect, you could also just type `remove-item c:\temp*`. That would be the most straightforward command, but it doesn't show how PowerShell differs from the regular command prompt.)

Here's another example of how a pipeline command can work:

```
dir | where-object {$_.LastWriteTime -lt <br>(get-date).addmonths(-6)} | remove-item
```

`dir` generates an object for each file in the current directory, `where-object` passes along only those that are more than six months old and `remove-item` deletes the old files. So, we've constructed a disk cleanup command out of general-purpose cmdlets. (Don't type this command, by the way—it's just an example!)

This is very peculiar and powerful, and it takes some getting used to.

An Extensible Environment

PowerShell can run three sorts of programs: built-in commands, external programs, and scripts. This is analogous to the regular Command Prompt environment, where you could use the built-in commands that are handled by the `cmd.exe` program itself, run external programs, or create batch files that let you orchestrate either type of command to perform the steps of a more complex task. In PowerShell, the built-in commands are cmdlets. Unlike the Command Prompt shell, however, these built-in commands are not wired into the PowerShell program but are added to it through a snap-in mechanism as one or more `.DLL` files stored on your hard disk. So, custom cmdlets can be added to the environment. The idea is that Microsoft and third parties can add install management cmdlets for their applications and servers, so that they can be managed by PowerShell scripts. Microsoft SQL Server, Microsoft Exchange,



Introduction to Windows PowerShell

and VMWare servers have custom cmdlet snap-ins, for example. If you're a skilled .NET programmer, you can create your own custom cmdlets using the Windows PowerShell 2.0 Software Development Kit (SDK) and Microsoft Visual Studio Express Edition, both of which you can download free from Microsoft.

In addition, PowerShell responds to command keywords that are part of its built-in script programming language which is a unique language, all its own. It's not VB-Script, it's not VB.NET, it's not C#. It's PowerShell.

You can also create and use .NET and COM objects not just in scripts, as you do with WSH, but right at the PowerShell command prompt.

This means that every cmdlet, Windows program, command-line program, .NET object, and COM object (including WMI and ASDI) is available right at your fingertips.

NOTE: As an aside, it's interesting to know that just as VBA provided an object-oriented, full-scale programming language that could be integrated into applications such as Microsoft Word, programmers can “host” PowerShell inside their .NET applications. PowerShell is intended not just for helping you automate your routine tasks, but to serve as the scripting/macro language for Windows applications and services.

Obtaining Windows PowerShell

Windows PowerShell 2.0 is installed by default on Windows 7 and Windows Server 2008 R2. For information on installing it on other versions of Windows, see [PowerShell details on the Microsoft Web site](#).

The PowerShell Environment

You should be able to click Start, All Programs, Accessories and see Windows PowerShell as a choice (although you might have to scroll down; by default, it's below System Tools). On 32-bit Windows versions, there are normally two choices:

- **Windows PowerShell**—This is the interactive command-line environment. I discuss this first.
- **Windows PowerShell ISE**—This is a GUI editing/debugging tool that you can use to help develop PowerShell scripts. I discuss this later.

On 64-bit versions of Windows, these two menu selections run the 64-bit version of Windows PowerShell. In addition, there are selections for Windows PowerShell (x86) and Windows PowerShell ISE (x86), which run the 32-bit versions of PowerShell. These are provided in case you have to use a cmdlet snap-in that is available only in a 32-bit version, but they are otherwise probably not of much interest.

Alternatively, you can open a regular Command Prompt window and type **power-shell** or type **powershell** in the Windows Search box or Run dialog box.

NOTE: If you run PowerShell inside a regular Command Prompt window, you won't get the default white-on-blue color scheme that you get when you open PowerShell from the menu, but it's the same program. You can return to the regular cmd.exe command prompt by typing **exit**.



Introduction to Windows PowerShell

Windows PowerShell is subject to the same security issues as the regular Command Prompt. To modify system files or perform other administrative actions on Windows 7 and Vista, you must run PowerShell with elevated privileges. On Windows XP, to perform privileged tasks, you must be logged on as a Computer Administrator or run PowerShell under an Administrator account using `runas`.

I discuss this in more detail toward the end of this article in the section [“PowerShell Security”](#)

Whichever method you use to start PowerShell, the next prompt you see should have the letters `PS` at the left, as shown in Figure 1 below. Type `dir` and press Enter, just to prove to yourself that this is not the old Command Prompt. The format of the directory listing is very different.

The `PS` to the left of the prompt tells you you’re using Windows PowerShell.

The PowerShell Command Prompt

Windows PowerShell prompts for commands just as the old Command Prompt environment does. It prompts, you type a command, you press Enter, it displays some sort of results, and you repeat the process over and over.

One thing that’s really improved with PowerShell, though, is that the command language is really a full-fledged programming language and creating, working with, and displaying variables is easy. For example, you might recall that you could use the `set name=xxx` command in batch files to store data in named variables and use the `%name%` syntax to retrieve the data.

In Windows PowerShell, you define variables by putting a dollar sign (\$) in front of the variable’s name to both define and retrieve its value. For example, you can type the commands

```
$a = 40 + 7 $a
```

and Windows PowerShell obediently displays the computed value of `a`, which is 47.

Every PowerShell command is just an expression with a value, so it’s perfectly legal to type `40+7` as a command line. PowerShell prints the result. These are trivial examples, but it can be quite powerful when you’re working with objects—just type the name of an object and its property to display the value of the property, for example:

```
$f.path
```

to print the `Path` property of an object you’ve put into variable `$f`.

Remember that whatever you can put into a script, you can also type directly at the PowerShell command prompt.

Command-Line Editing

PowerShell lets you use the backspace, arrow, and other editing keys on your keyboard to modify command lines as you’re typing them and to recall, modify, and reuse previously typed commands. This will be very familiar if you’ve worked with the Command Prompt because the editing keys are identical. (There’s a reason for this: PowerShell is a standard console application just like `cmd.exe`.)



Introduction to Windows PowerShell

The arrow key functions are the most important to remember:

- The left and right arrow keys (← and →) let you move within a command line that you're typing, so you can go back and change a mistake at the beginning of the line without erasing and retyping the whole thing.
- Ctrl+← and Ctrl+→ move back and forth a whole word at a time. This can save a lot of keystrokes.
- The Home and End keys move instantly to the beginning or end of the line.
- Normally, if you move backward in the edited line and start typing, what you type is inserted ahead of what's already there. This is usually very handy. If you need to replace a lot of text, though, instead of erasing what's there and then typing in the new, press the Ins key once and your typing replaces what's already there—just type over it. The cursor changes to a solid block to remind you that this is what will happen when you type. You can press Ins again to return to Insert mode. PowerShell automatically switches back to Insert mode when you press Enter, so you can edit the next command normally.
- The up and down arrow keys (↑ and ↓) let you step back to commands you typed previously. You can use ↑ to locate a previous command and press Enter to reissue it, or you can locate it and then edit it before pressing Enter.

There are some other function keys that you might want to know about. These shortcuts actually work in *any* Powershell or Command Prompt window, and with any console application:

KEY	DOES THE FOLLOWING...
F2	F2 followed by a single character copies text from the previously entered command line into the current command line, up to but not including the first occurrence of the character you type. For example, if the previous command was <code>get-childitem c:\temp get-member</code> then pressing F2 copies <code>get-childitem c:\temp</code> into the current input line.
F3	Types in whatever was in the previous command line from the cursor position to the end of the line. For example, if you mistype a single character in a command line: <code>fet-childitem c:\temp</code> and don't realize it until you've pressed Enter, just type <code>g</code> and then press F3 to recall the rest of the line. You'll end up with <code>get-children c:\temp</code> .
F4	F4 followed by a single character deletes text from the cursor up to but not including the first occurrence of that character.
F7	Pops up a list of recently typed commands. You can scroll through it using the arrow keys and press Enter to reissue the command.
F9	Lets you type in the number of a previous command. The numbers are the ones shown in the F7 pop-up. (This one would have been useful if the numbers ran backward, so that 3 meant the third command back, but they don't, so it's not.)





Introduction to Windows PowerShell

To be honest, you can work through an entire IT career without ever using these F-key shortcuts and not suffer for it, but if you can manage to remember them, they could come in handy some day. I keep meaning to remember F7, but I never do.

Copying and Pasting

Copying and pasting in the PowerShell environment works exactly as it does in any Command Prompt or console program window, again, because PowerShell is a normal console program, just like cmd.exe.

You can't cut text out of the window; you can only copy text from a rectangular region to the clipboard. To copy from the PowerShell screen with the mouse, click the upper-left corner of the PowerShell window and select Edit, Mark. Point to the upper-left corner of the text you want to copy, press the mouse button, and drag to the lower-right corner of the text you want. Then, right-click the screen, or press Enter, to copy the text.

NOTE: If all the text you want isn't visible, remember that you can scroll the window up and down during the process.

There are also keyboard shortcuts for these steps. Press Alt+Space E K to begin. You can then use the arrow keys to move the cursor to the upper-left corner of the desired text; then, hold down Shift and while holding it, use the arrow keys to move to the lower corner. Press Enter to complete the operation.

Personally, I find it easiest to type Alt+Space E K to start, switch to the mouse to mark the rectangle, and then go back to the keyboard to press Enter.

To paste text from the clipboard into the PowerShell window, click the upper-left corner of the window and select Edit, Paste. You might find it quicker to use to the keyboard shortcut: Alt+Space E P.

Pausing Output and Stopping a Runaway Program

As in the regular Command Prompt environment, you can type Ctrl+C to cancel a running PowerShell program or script.

In addition, Ctrl+S stops a printout that's scrolling by so fast you can't read it, and another Ctrl+S starts things up again. You can also use the scrollbar to look back through the contents of the PowerShell window.

Still, if a command is going to generate a lot of output, just as in the Command Prompt environment, it's best to either pipe the output through the `more` command, which pauses after every screenful, or direct the output into a file, and then view the file with Notepad or a word processor.

Command-Line Syntax

PowerShell's command-line syntax is at once familiar and a bit strange. I discuss this in the following sections.



Introduction to Windows PowerShell

Command Types

As I mentioned previously, PowerShell has built-in language statements, and it can run internal cmdlets, external commands, and scripts that contain multiple PowerShell commands.

With the exception of the PowerShell language keywords like **if** and **foreach**, which can have complex structures, commands are evaluated one line at a time, just as in the Command Prompt world.

The first word on the line is the name of the command you want to run. Any remaining text on the line is passed to the command as its arguments. Normally, spaces delimit the command name and the arguments, but you can use the single or double quotation mark (") character to embed a space in an argument, as in the following command:

```
command /abc "argument with spaces"
```

In an unusual twist, to type a program name that has spaces in its name or path, you must use single or double quotes around the name and must precede the command with an ampersand, as in this example:

```
&"command with spaces" /abc argument
```

PowerShell finds the program that corresponds to the command name by searching the following places in this order:

1. PowerShell first looks through its alias list, which is a list of shorthand names for commands. PowerShell is installed with nearly 150 built-in aliases predefined. For example, **cd** is set up as the alias for the **Set-Location** cmdlet. As you work with PowerShell, you'll find that you'll soon want to define your own custom aliases for the commands you use most often.
2. If an alias name is matched, PowerShell replaces the command name in your command line with the alias definition and the search process over.
3. PowerShell then scans the names of functions that have been defined by the current script, if you're running a PowerShell script, or that have been defined at the command line. If a function with the specified name is found, the function is called and passed the command-line arguments. You can define your functions in the PowerShell language in scripts or at the command prompt. They work and act just like cmdlets.
4. If the command name is not found in the alias list, PowerShell searches the list of cmdlets that come preinstalled with Windows PowerShell or have been added by a snap-in that came with an added application or service. If it's found, the cmdlet is run.
5. If the name doesn't match a cmdlet, PowerShell uses the standard search path mechanism to look for an external program, much as cmd.exe does. It looks first in the current directory and then through the directories listed in the **PATH** environment variable for a file with the stated command name and whose extension is a recognized program extension. In each folder searched, PowerShell does the following:



Introduction to Windows PowerShell

- It first looks for the extension `.ps1`, which is used for PowerShell scripts in PowerShell versions 1.0 and 2.0. If a script file is found, PowerShell runs the script (subject to the security restrictions that I discuss shortly).
 - If a `.ps1` file is not found, it scans through the extensions listed in the `PATHEXT` environment variable, just as `cmd.exe` would. This mechanism lets it find and run standard Windows executable files (`.exe` files), both GUI and console applications. `PATHEXT` also includes `.cmd` and `.bat`, so that PowerShell can run normal Command Prompt batch files—it fires up a copy of `cmd.exe` to do so. Likewise, for `.vbs`, `.js`, `.vbe`, and `.jse` files, it fires up `WSH` to run these scripts, and for `.MSC` files, it runs the snap-in through the Microsoft Management Console.
6. Thus, you can run PowerShell cmdlets and scripts, console applications, Windows programs, batch files, or any other sort of program from the PowerShell environment. `notepad` still starts Notepad.

There are exceptions to the normal search processing, though:

- If your command includes an explicit path as part of the command name, PowerShell runs the specified file. For example, the command `c:\bin\sort.exe` would run that specific copy of `sort.exe` and would not try to search for `sort` as an alias or in the `PATH` locations.
- To improve security, PowerShell will not run a `.ps1` PowerShell script that it found in the current directory before it searched the `PATH` list. It will tell you about it, but it won't run it. It will run only PowerShell scripts that it finds in the `PATH` or those to which you've explicitly typed a path.
- You can run a file in the current directory by preceding its name with `.\`, as in `.\myscript.ps1`. This is an explicit path, so it satisfies PowerShell's security requirement.
- Even then, PowerShell might not run scripts at all, depending on additional security settings that I discuss in the next section.
- It's conceivable that more than one PowerShell snap-in might define a cmdlet with the same name. PowerShell searches through snap-ins from the most recently installed back toward the first installed, and the first snap-in found that defines the cmdlet name is the one that's used. Also, you can import (load) new snap-in cmdlet modules while PowerShell is running, and the same ordering applies: Snap-ins are examined from the most recently imported toward the oldest.
- Or, you might have defined an alias that has the same name as an existing cmdlet. If you type the aliased name, you'll get the command that the alias specifies, not the original cmdlet.
- To run a specific cmdlet from a specific snap-in without hitting this ordering or aliasing problem, you can precede the cmdlet name with the name of the snap-in module, followed by a backslash. For example, `Microsoft.PowerShell.Utility\Get-Date` runs the `Get-Date` cmdlet in the `Microsoft.PowerShell.Utility` module even if you've defined an alias for `Get-Date` or another module has defined a cmdlet with the same name.



Introduction to Windows PowerShell

- For more information on command searching, type `help about_command_precedence` at the PowerShell command prompt.

I know this can seem tedious in an introduction, but you need to know the details of the search system to get your own scripts to run and to diagnose problems if someday the program that runs isn't the one you expected.

The main thing you should take from this is that if you want to develop and use your own PowerShell scripts, you will want to create a specific folder to hold them and put that folder name into the Windows PATH environment variable. If you've already set up such a folder for batch files or WSH scripts, you can use the same folder for PowerShell scripts.

Redirection and Pipes

You can redirect the output of commands you run from the PowerShell prompt and in PowerShell scripts into files. The usual operators—`|`, `>`, `<`, `>>`, and so on—work as they do in the Command Prompt environment.

In PowerShell, though, remember that most cmdlets emit objects, not text. The output of these cmdlets turns into text listings only when they land in the PowerShell window, are redirected to files, or piped to standard Windows console commands. You can control the formatting of these text listings using specific cmdlets.

Cmdlets and Objects and Scripts, Oh My!

The PowerShell world introduces new, somewhat confusing concepts and terminology, and I suspect that your first reaction will be similar to mine: I looked at a few PowerShell scripts; scratched my head; and thought, "I wonder if there are any donuts left in the kitchen?" At first, it all seemed pretty opaque and possibly not worth investigating. Fortunately, this feeling didn't last long. (Neither did the donuts.)

The biggest obstacles are the strange names of the cmdlets and the unusual command-line syntax. Cmdlets have names like `Remove-ItemProperty`, `Get-Content`, `Wait-Job`, `New-Alias`, and so on. This can help make sense of it:

- Cmdlet names are case insensitive. You can type `wait-job` just as well as `Wait-Job`. The capitalization is used to make the words more distinct when you read the documentation.
- Microsoft's programmers chose to use a noun-verb convention for naming the cmdlets to make the names more clearly say what the cmdlets actually do. The dash is just part of the name. `Rename-Item` isn't some sort of combination of a `Rename` command and an `Item` command. The name is all one word: `Rename-Item`, and you can guess from this name that it probably renames whatever items it's given.

Yes, the names tend to be long and cumbersome to type. A Unix programmer would have chosen a name like `ri` for that `Rename-Item` command. But, `ri` doesn't really tell what it does. As it turns out, PowerShell does have a built-in alias for `Rename-Item`,

Introduction to Windows PowerShell

and the alias is...ri. As you learn PowerShell's commands, you might start by using the long names to help remember what the commands do and then scan through the alias list and start learning their shorter abbreviations.

- The names of cmdlet command-line options are similarly long. For example, the `get-process` cmdlet can take the following optional arguments:

```
-Name "string"<br> -ComputerName "string"<br> -FileVersionInfo<br> -Module<br> -Id integer<br> -InputObject object
```

Again with the long names! Who can type `-InputObject` quickly without making three typos along the way? Well, it turns out that you don't have to type the entire option name, just enough to distinguish the desired option from any others that start with the same letters. So, you can type `-input` or even `-in` instead of `-InputObject`. You just can't shorten it to just `-i` because that's not enough to distinguish between `-Id` and `-Inputobject`.

Some command can be shortened even more. For example, a typical `new-alias` command looks like this:

```
new-alias -name shortname -val realcommandname -descr "Brief description"
```

But, you even can omit the parameter switches, in this really short version of the command:

```
new-alias shortname realcommandname
```

The two arguments in this case are called *positional parameters* because PowerShell has to figure out what they are based on their position in the command line. You can see which parameters can be specified this way by looking at a cmdlet's syntax description. If you type `new-alias -?`, you can see that the syntax is

```
<strong>New-Alias [-Name]</strong> <em><string></em> <strong>[-Value]</strong>  
<em><string></em> <strong>[-Description</strong> <em><string></em><strong>]</  
[-WhatIf]</strong> [<em><CommonParameters></em>]
```

Notice that the `-Name` option indicator itself is listed in `[]` brackets, so it's optional, but the `<string>` value after it isn't. This argument must always be there, so it can be interpreted by its position. The `-Value` string is listed the same way, but none of the other arguments are.

The other main obstacle to learning to use PowerShell is the strange syntax with commands like the example I gave earlier

```
dir | where-object {$_.LastWriteTime -lt (get-date).addmonths(-6)} | remove-item
```

The `dir` and `remove-item` parts are self-explanatory, but the middle part is, well, non-obvious. This where you hit the PowerShell learning curve, and the only thing to do is expose yourself to a lot of it. It will start to make sense pretty quickly. My advice

Introduction to Windows PowerShell

is to read as many PowerShell scripts as you can. Read them like literature, get a feel for the language, and work out what they do. I'll just point out a couple of things about that example command here.

The `where-object` cmdlet is a filter. It reads a stream of objects from the previous command in the pipeline, evaluates a true/false value for each one, and passes along only the objects for which the value is true. In the example, the true/false value is provided by the part within curly brackets:

```
{$_ .LastWriteTime -lt (get-date).addmonths(-6)}
```

CAUTION: It's important to know that this expression isn't handled directly by `where-object`. If it was, then conceivably every cmdlet might have a different way of constructing these expressions, which would make things inconsistent and difficult to master.

Instead, this expression format is part of the PowerShell language itself. Technically, `where-object` expects the body (code) of a function on its command line. The curly brackets signify that you're typing the body of a function that you're defining on-the-fly without giving it a name, and that function is given to `where-object` to use to evaluate each of the file objects it scans. (This little function is technically called a lambda, but that's not too important here.)

Let's pick it apart. There are three parts:

<code>\$_ .LastWriteTime</code>	<code>\$_</code> is a placeholder for the object that <code>where-object</code> is examining. <code>\$_ .LastWriteTime</code> picks up the <code>LastWriteTime</code> property from that object. For a file or folder object, it's the date and time that the file was last modified.
<code>-lt</code>	This does a comparison of two values, and it's true if the first value is less than the second. In other words, the expression <code>{a -lt b}</code> , is true if value <code>a</code> is less than value <code>b</code> . In this instance, the test is true if the file's <code>LastWriteTime</code> is less than (earlier than) the time that comes after <code>-lt</code> .
<code>(get-date).addmonths(-6)</code>	This is the oddest part, but watch how it works: The <code>get-date</code> cmdlet is run. With no arguments, it spits out a <code>.NET System.DateTime</code> object representing the current date and time. In this simplest form, it's like <code>Now()</code> in VBScript. The parentheses around it indicate that we are going to treat the result of the cmdlet as a value. <code>addmonths</code> is one of the standard methods of the <code>System.DateTime</code> object. In this case, we end up with the current date with six months subtracted from it—in other words, the date six months back.

The end result is that this expression is true if the file was last written more than six months ago. The curly brackets turn this expression into a function so that `where-object` can evaluate it for each of the objects it examines.

The output of this `where-object` command is just those file objects that were last



Introduction to Windows PowerShell

modified more than six months ago. The rest of the command line feeds that stream of objects to `remove-item`, which deletes the files they represent. The end result is that this command purges a directory of old, abandoned files.

Now, I didn't just write that command off the top of my head. I had to learn the basics of cmdlets first, so I knew to write:

```
dir | where-object {some expression} | remove-item
```

Then, I had to learn about the properties of the file and folder objects to know to use `LastWriteItem` and the `System.DateTime` object to find the `addmonths` property. To help with that job, interestingly enough, there are cmdlets that describe the properties of any object they encounter. I describe that next.

Getting Help

The most important PowerShell command to learn first is the one that helps you learn about the others! It's called `get-help`, but it has a predefined alias named `help`, so you can use either of these command names interchangeably. PowerShell has a lot of built-in help—477 different topics, in fact, at the time I wrote this.

Here are some tips:

- Type the word `help` by itself to get a quick introduction to the online help system.
- After `help` or `get-help`, you can type a word, cmdlet name, or partial cmdlet name. You can also type a phrase enclosed in quotation marks. If more than one topic, cmdlet name, or description matches what you typed, `get-help` prints a list of all matching help entries. You can then type `help` followed by the specific entry name that interests you to get the details. If exactly one name or description contains the word or phrase you typed, `get-help` prints the help information for that topic.
- Some cmdlets have additional help information available. For example, you can see some examples of using the `new-alias` cmdlet by typing `help new-alias -examples`. Common options for additional information are `-examples`, `-detailed`, and `-full`. The main help page for a topic tells you whether these expanded pages are available.
- The help text appears in the console window and by default is piped through `more` so it stops after every screenful. Press Enter to go on to the next page.
- Alternatively, you might find it easier to direct help output into a file by adding, for example, `>x` to the end of the help command line; then, type `notepad x` to read the text.
- There are help entries covering every one of the installed cmdlets and additional articles on different topics. These articles start with `about_`. For example, `help about_execution_policies` prints information about the scripting security restriction system. (You can also type `help about_exec` and get the same result because only one entry contains the string “about_exec” in its title or description.) To see a list of all of these “about” entries, type `help about`.

This online help system is great for locating cmdlets based on words in the description of the job they do and for learning about their command-line syntax after you've found a cmdlet that interests you.

Introduction to Windows PowerShell

TIP: In the online syntax descriptions, the following conventions are used:

[] square brackets indicate optional command arguments.

{ } curly braces usually indicate a series of options from which you can choose, with a vertical bar | between the options.

< > angle brackets surround a value that you have to supply yourself. For example,

[-Description <string>] indicates an optional argument. You can omit it, or you could type something like **-description "some text"**. As I mentioned in the previous section, you could probably abbreviate this to **-descr "some text"**.

Prompting to Complete Commands

If you type a PowerShell command that requires specific arguments, but omit some or all of the required arguments, PowerShell prompts for them by name. Here's an example: If you type **set-alias** without any arguments, PowerShell prompts for the required **-name** and **-value** arguments. Here's how it looks:

```
PS C:\Users\bknittel> set-alias <br> <strong>cmdlet Set-Alias at command pipeline  
position 1</strong> <br> Supply values for the following parameters: <br> Name:  
<em>ax</em> <br> Value: <em>get-alias</em>
```

(I typed the responses in italics.) This mechanism can help you as you learn to use the cmdlets, but remember that PowerShell won't prompt for optional arguments, just the required ones.

Personally, I don't find this mechanism that useful and never trigger it deliberately. You should know it exists though because if you omit a required argument from a command, PowerShell might prompt you in this way and you might find it confusing at first.

Aliases

As I mentioned previously in "Command-Line Syntax," when you type a command, PowerShell looks through list of aliases, functions, cmdlets, and external programs to find the program you want to run.

Because people who do a lot of work at the command prompt are so used to typing commands like **cd** and **dir**, PowerShell comes with predefined aliases for many of these familiar commands. Because the names of many of the cmdlets are tediously long, there are predefined aliases for many of the cmdlets that are only two or three characters long. (This should make Unix users happy!) The idea is that you'll find that there are just a few commands you tend to use over and over. After you know which these are, you can look up their short aliases and use those to save time typing.

One serious limitation of aliases, though, is that they can map only one command name to another. An alias definition can't include command-line arguments. I would really like to define an alias named **h** that issued the command **cd \$home**, which changes back to your user profile "home" directory, but this isn't supported. It's really unfortunate.



Introduction to Windows PowerShell

How to Get a Listing of Aliases

You can see a list of all the built-in aliases by typing `alias` (which, as you'll see, is an alias itself, for the real command name `get-alias`). I find it easiest to view this list in a text file by typing two commands:

```
alias >x <br> notepad x
```

You'll notice the alias named `%`, for `ForEach-Object`, and the alias named `?`, for `Where-Object`. The latter, used in that “delete old files” command example I gave earlier in the chapter, would look like this:

```
dir | ? {$_.LastWriteTime -lt (get-date).addmonths(-6)} | remove-item
```

Personally, I don't think it reads well, but if I got used to typing it, it might save some time. It's up to you whether you want to use shortcuts like this, so nobody's forcing anything on you. The idea is to use the shortcuts that make sense to you and ignore the others.

How to Define a New Alias

You can define a new alias by typing a command like this:

```
new-alias -name <em>shortname</em> -value <br><em>realcommandname</em>  
-description <em>“Brief description”</em>
```

but substituting the name of the alias you want to create for *shortname* and the command that the alias should run for *realcommandname*. If the alias already exists, you have to add `-force` to the command to replace the old one. I show you how you can shorten this command later in this article.

You should know that alias definitions don't survive when you close your PowerShell window. The next time you run PowerShell, your custom alias will be gone. If you come up with favorite aliases you want to keep, see the section on Profiles at the end of this article.

Navigating Directories and Other Locations

In the command-line world, you use typed commands to navigate up and down through folders and disk drives. Windows PowerShell uses the same `cd` command as the Command Prompt to change directories, so familiar commands like

```
cd \Users <br> cd .. <br> cd subfolder
```

still work. You can use the Tab key to automatically fill in partially typed file and folder names, just as in the Command Prompt environment.

You can also take advantage of directory names stored in PowerShell variables. Your user profile folder path is stored in variable `$home`, so you can use the command `cd $home` to return to your profile directory from any location.

Now, here's something really interesting: You're used to using `cd` to move around the file system. In PowerShell, you can also navigate through the Registry!

Here's how it works: Just as Windows recognizes different drive letters, like this:

Introduction to Windows PowerShell

```
c:\users\bknittel <br>  
d:\saved_pictures\February2010\Yosemite <br>  
e:\setup.exe
```

PowerShell recognizes additional drives, which it calls providers, that treat the Registry and the lists of environment variables, aliases, PowerShell variables, defined functions, digital certificates, and Web Services for Management just like disk drives. You can specify paths to these elements, list them with `dir`, and in some cases even use `cd` to navigate through them.

The default provider drives on Windows 7 are listed in Table 1.

Table 1: Provider Drives on Windows 7

DRIVE NAME	CONTAINS
A:, B:, C:, and so on	Disk drives (FileSystem provider)
Alias:	List of PowerShell aliases
Cert:	Digital certificates
Env:	Environment variables
Function:	Defined functions
HKCU:	Registry section HKEY_CURRENT_USER
HKLM:	Registry section HKEY_LOCAL_MACHINE
Variable:	PowerShell variables
WSMan:	Windows Services Management connections to the local and remote computers

Name completion works with these drives, too. For example, if you type `dir hklm:soft` and press the Tab key, PowerShell changes `soft` to `SOFTWARE`, which is the first Registry key that matches the partial name you typed. You can continue the path by typing `\` and another part of a key name.

Try typing these commands at the PowerShell prompt (and notice that you don't have to capitalize them; the names are case insensitive):

<code>dir hklm:\software</code>	This lists the keys under HKEY_LOCAL_MACHINE\SOFTWARE.
<code>cd hklm:\software</code>	Makes the Registry your "current location."
<code>dir</code>	Lists the keys in this current location.
<code>cd \$home</code>	Returns to the file system in your profile folder.

Introduction to Windows PowerShell

<code>dir cert:\currentuser</code>	Lists certificates associated with your user account.
<code>dir variable:</code>	Lists all variables currently defined in Windows PowerShell.
<code>dir env:</code>	This lists all defined environment variables.

Many PowerShell cmdlets can work with objects specified by their paths whether the paths point to a file, a Registry key, a certificate, or whatever. For example, the `del` command (which is an alias for `delete-item`) can delete a Registry key just as easily as a file, with a command like `del hkcu:\software\badkey`.

PowerShell Security

Because Windows PowerShell can be used to change Windows settings and has the potential, if run by a privileged user, of undermining Windows' security system, Microsoft has taken care to be sure that it's difficult to run PowerShell scripts without your taking intentional steps to enable them. There are additional barriers to running scripts that come from the Internet, via email, Instant Messaging programs, and so on. This was done to avert the possibility that hackers might figure out ways to exploit potential flaws in, say Internet Explorer, to install and run PowerShell scripts on your system without your permission.

PowerShell Scripts and User Account Control

Windows PowerShell was designed first and foremost for managing Windows, so you will end up wanting it to do things that require Administrator privileges. On Windows 7 and Vista, PowerShell, requires elevated privileges—just running it from a Computer Administrator account isn't enough. Either open an elevated command prompt and then type `powershell` or use any of the methods you'd use to run `cmd.exe` with elevated privileges, but run `powershell.exe` instead.

On Windows XP, to perform administrative functions, you'll have to be logged in as a Computer Administrator when you start PowerShell. Alternatively, you can use the `runas` command to start `powershell.exe` in the context of an Administrator's user account.

Whatever version of Windows you're using, you'll probably want to try this right away, as you want to use a privileged PowerShell command to let you run PowerShell script files. I describe this command next.

Script Execution Policy

As initially installed, PowerShell is set up so that it can be used interactively at the command prompt but will not run scripts. This does make sense. The majority of Windows users will never touch PowerShell, and nobody knows what future bug might be discovered that would let hackers silently send them malicious PowerShell scripts via Internet Explorer, Adobe Acrobat, YouTube, or who knows what. Better to be safe than sorry.

Power users just have to change the default script execution policy from Restricted

Introduction to Windows PowerShell

to, as I recommend, RemoteSigned. Table 2 lists the possible policy settings.

Table 2 PowerShell Script Execution Policy Settings

SETTING	DESCRIPTION
Restricted	This is the default setting. No PowerShell scripts will run at all, under any circumstances.
AllSigned	Only digitally signed scripts (including profile scripts, which I describe in the next section) will run; in addition, you are prompted to grant permission to run scripts signed by the specific certificate used.
RemoteSigned	Scripts (and profiles) that have been written locally will run. Scripts that have been downloaded from the Internet will not run unless they are signed and you have approved the signing certificate.
Unrestricted	All scripts will run, but you are warned about scripts that have been downloaded and must approve them before they'll run.
Bypass	Any script will run, regardless of its origin. This is a potentially very risky setting and should be used only in very specific circumstances, where some other security system is in place to prevent rogue scripts from running without your permission.
(Undefined)	If no policy setting is in place, PowerShell treats the setting as Restricted and no script will run.

I recommend the RemoteSigned setting. Scripts you write and store on your computer or retrieve across your local area network will run without any problem, but scripts that arrive via programs with Internet connectivity won't run unless they have a digital signature and you've confirmed that you trust the person or company that signed the script. This is a good security compromise.

To change the setting, open a privileged PowerShell window as described in the previous section. Then, type the command

```
set-executionpolicy remotesigned
```

You will have to confirm that you want to make this change by typing *y* and Enter. (You can disable that prompt by adding *-force* to the command line.)

Now, you can run PowerShell scripts and profiles. .

There are few things you should know about this security setting:

- If your computer is part of a corporate domain network, this setting is probably under the control of your network manager via Group Policy. Type `get-execution-policy` to see what your current setting is. You might not be able to change it yourself.
- If you are forced into, or choose, the AllSigned policy setting, you'll have to digitally sign any script you want to run. It's a cumbersome process, but it's not too bad after you get used to it.



Introduction to Windows PowerShell

- You might recall that files downloaded from the Internet are marked as “potentially risky” through information stored in a separate data stream associated with the downloaded file. To remove this marker from a downloaded script that you are certain is safe, use Windows Explorer to open the file’s properties dialog box and click Unblock. It will now be considered a local file and will run under the RemoteSigned execution policy.

Finally, remember that if you have a local area network (LAN) but not a Windows domain network, and the scripting execution policy isn’t set by Group Policy, you’ll have to change this setting on every computer on your network that you want to manage using PowerShell.

PowerShell Profiles

As I showed in the previous sections, you can customize the PowerShell environment to suit your preferences by adding custom aliases, adding directories to the path, and so on. It would be a pain to have to retype these commands every time you started PowerShell—and you don’t have to if you set up a PowerShell profile, which is a script of commands that PowerShell runs every time you start a new instance. You might want to put commands of this sort in a profile script:

```
new-alias -name “n” -val<br> “notepad” -descr “Edit a file with Notepad”<br> $env:path  
+= “c:\PSscripts”
```

so that your favorite, custom aliases will be available every time you use PowerShell.

Here’s the skinny: Whenever you start Windows PowerShell in any of its various flavors (the regular command-line PowerShell, the GUI PowerShell ISE, or a PowerShell variant that’s embedded inside another application), it looks for profile scripts in the following two locations, in this order:

```
%windir%\system32\WindowsPowerShell\v1.0
```

where %windir% is the folder in which Windows is installed, usually c:\windows, and

```
%userprofile%\[My] Documents\WindowsPowerShell
```

where %userprofile% is your account’s profile folder, usually c:\Users\username on Windows 7 and Vista and c:\Documents and Settings\username on Windows XP.

Within these two folders, the different flavors of Windows PowerShell look for different profile filenames. Every PowerShell variant looks first for a script named `profile.ps1` and runs it if it’s found. The command-line PowerShell then looks for a profile script named `Microsoft.PowerShell_profile.ps1` and executes that one if it’s found. The GUI PowerShell ISE program, on the other hand, looks first for `profile.ps1` and then `Microsoft.PowerShellISE_profile.ps1`.

Profile scripts in the %windir% folder affect all users on the computer, while profile scripts in your %userprofile% location affect only shells that you use. So, you can decide to make some customizations available to everyone and make some just for



Introduction to Windows PowerShell

yourself by setting up separate profile scripts. Likewise, you adjust every PowerShell variant by putting commands in `profile.ps1`, or you can season the different PowerShells to taste by putting commands in the version-specific files.

PowerShell executes each of the scripts it finds, so the environment you end up with is the result of the cumulative changes made by all the script(s) that you've set up.

Creating a PowerShell Profile: PowerShell has a predefined variable named `$profile` that points to the version-specific profile file in your `%userprofile%` folder. For example, in my command-line PowerShell, `$profile` has the value "`c:\Users\bknittel\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1`".

So, you can instantly create a PowerShell profile by typing the command `notepad $profile`. Notepad will ask whether you want to create a new file. Click Yes, and you're ready to go.

To create a profile for all users in folder `%windir%\system32\WindowsPowerShell\v1.0` on Windows 7 or Vista, remember that you'll need to use an elevated Command Prompt or PowerShell instance. This folder name is stored in variable `$PSHOME`, so from PowerShell you can use the command `notepad $psHOME\profile.ps1` or `notepad $psHOME\Microsoft.PowerShell_profile.ps1` to create or edit the file.

However, and this is a big *however*, profile scripts are still scripts and PowerShell will only run profile scripts that meet the security guidelines I described in the previous section. Remember that the default execution policy is Restricted, so profile scripts won't run at all (unless you're on a Windows domain network and your network manager has enabled script execution).

To take advantage of the profile feature to customize your PowerShell environment, you will need to change the execution policy and possibly digitally sign your scripts. The previous section tells you how to do this.

This article is excerpted from the book [Windows 7 and Vista Guide to Scripting, Automation, and Command Line Tools](#), copyright Pearson Education, all rights reserved. Reprinted with permission.

Getting started with PowerShell: The basics

Learn how to perform simple tasks and make your way around the syntax. By Jonathan Hassell

ARE YOU a Windows administrator? Did you make a new year's resolution to learn PowerShell in 2015? If so, you have come to the right place. In this piece, I will get you started by orienting you to the world of PowerShell, helping you get your bearings and showing you how to perform simple tasks with the language so that you have a solid foundation on which to add skills for your particular job. Let's get started.

Get-Basics

PowerShell uses a consistent syntax for all of its commands — in fact, PowerShell commands are actually called cmdlets, because they're much more than simple DOS-style actions. All cmdlets use the following syntax:

Verb-Noun

You can easily remember it as “do something to” “this thing.” For example, here are three actual cmdlets:

- Set-ExecutionPolicy
- Get-Service
- Show-Command

All cmdlets will always follow this format. Using these three, you will set an execution policy, show a command or list of commands and get some information about a service and what it can do.

There are a few things to remember about using PowerShell at any time:

- **PowerShell is case-insensitive.** UPPERCASE, lowercase, cAmElCaSe — it doesn't matter. PowerShell simply reads the text in and performs the action you want.
- Since PowerShell cmdlets are always consistently formatted, you can **chain those cmdlets and their output together and do things in sequence.** For example, one cmdlet can retrieve a list of things, and you can send that list (the output from that first command) to a second command, which then does things too. This can go on and on and on as long as you need it to until whatever task you want is complete.
- **The output of a PowerShell cmdlet is always a .NET object.** This might not mean a lot to you right now, especially if you are not a programmer or don't have a software development background, but you will find as you learn more about the PowerShell language that this is where some of the real power in PowerShell lies.

Introduction to Windows PowerShell

- PowerShell works well with local systems but it **also works tremendously well “across the wire” with remote systems**. With a few changes to settings on both your local machine and the remote systems you want to manage, you can execute commands across thousands of machines at one time. This had been the purview of Bash and Perl before PowerShell, but now Windows has a true remote command shell equivalent.

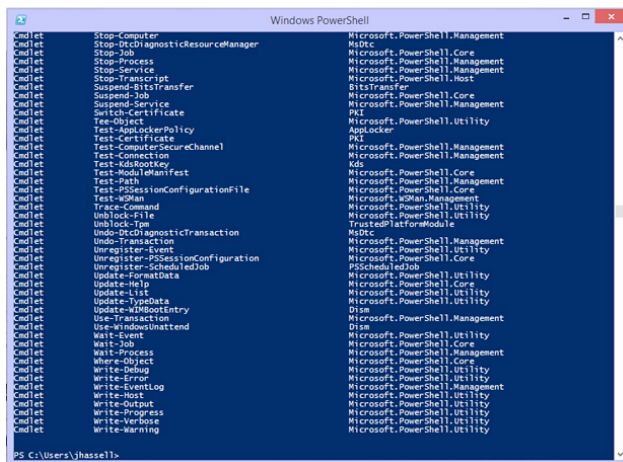
Seeing the universe

On a machine where you have PowerShell installed, open it up and just type in the following:

Get-command

This tells PowerShell to get all of the cmdlets it knows about. In the absence of anything else, PowerShell will always send output to the screen. (You could direct this to a file, into another PowerShell cmdlet or use it some other way, but for now let's just get a sense of what can be done.)

You should see something like this:



```
Windows PowerShell
cmdlet Stop-Computer Microsoft.PowerShell.Management
cmdlet Stop-DiagnosticResourceManager MsdC
cmdlet Stop-Job Microsoft.PowerShell.Core
cmdlet Stop-Process Microsoft.PowerShell.Management
cmdlet Stop-Service Microsoft.PowerShell.Management
cmdlet Stop-Transcript Microsoft.PowerShell.Host
cmdlet Suspend-BitsTransfer BitsTransfer
cmdlet Suspend-Job Microsoft.PowerShell.Core
cmdlet Suspend-Service Microsoft.PowerShell.Management
cmdlet Switch-Certificate PKI
cmdlet Tee-Object Appender
cmdlet Test-AspLockerPolicy PKI
cmdlet Test-Certificate Microsoft.PowerShell.Management
cmdlet Test-ComputerSecureChannel Microsoft.PowerShell.Management
cmdlet Test-Connection Kds
cmdlet Test-ModuleManifest Microsoft.PowerShell.Core
cmdlet Test-PATH Microsoft.PowerShell.Management
cmdlet Test-PSessionConfigurationFile Microsoft.PowerShell.Core
cmdlet Test-SPMan Microsoft.SDMan.Management
cmdlet Trace-Command Microsoft.PowerShell.Utility
cmdlet Unlock-File Microsoft.PowerShell.Utility
cmdlet Unlock-Type TrustedListForModule
cmdlet Undo-DiagnosticTransaction MsdC
cmdlet Undo-Track Microsoft.PowerShell.Management
cmdlet Unregister-Event Microsoft.PowerShell.Utility
cmdlet Unregister-PSessionConfiguration Microsoft.PowerShell.Core
cmdlet Unregister-ScheduledJob PSScheduledJob
cmdlet Update-FomatData Microsoft.PowerShell.Core
cmdlet Update-Help Microsoft.PowerShell.Utility
cmdlet Update-TypeData Microsoft.PowerShell.Utility
cmdlet Update-WebResourceEntry Dism
cmdlet Use-Transaction Microsoft.PowerShell.Management
cmdlet Use-WindowInattend Dism
cmdlet Wait-Event Microsoft.PowerShell.Utility
cmdlet Wait-Job Microsoft.PowerShell.Core
cmdlet Wait-Process Microsoft.PowerShell.Management
cmdlet Where-Object Microsoft.PowerShell.Core
cmdlet Write-Debug Microsoft.PowerShell.Utility
cmdlet Write-Error Microsoft.PowerShell.Utility
cmdlet Write-EventLog Microsoft.PowerShell.Management
cmdlet Write-Host Microsoft.PowerShell.Utility
cmdlet Write-Object Microsoft.PowerShell.Utility
cmdlet Write-Progress Microsoft.PowerShell.Utility
cmdlet Write-Verbose Microsoft.PowerShell.Utility
cmdlet Write-Warning Microsoft.PowerShell.Utility
PS C:\Users\jhasell>
```

PowerShell's Get-command cmdlet.

That's the universe of PowerShell — at least to your machine right now. Other products like Exchange, SharePoint, System Center and Office 365 all have their own set of cmdlets that you would need to import to do their service-specific tasks, but let's just focus on what your machine knows about now.

That's still a lot of cmdlets! How many, you ask? Well, let's figure out how to get PowerShell to tell us. Remember how I said that the output of a PowerShell cmdlet is always a .NET object? Well these objects all have properties — exactly the same way that Windows files have properties. For example, when you right-click on a file in Explorer and click Properties, you get what amounts to a bunch of attributes and values



Introduction to Windows PowerShell

describing the file — author, location, keywords, permissions and so on.

Well, .NET objects have properties too, and one of the properties you get right out of the box is the count property. At its simplest, the count property can tell you the number of different things you're dealing with — users, mailboxes and so on. You can simply grab the value of the count property of any cmdlet's output by enclosing the cmdlet in parentheses, including a dot, and then putting in the name of the property, count. Like this:

```
(get-command).count
```

Remember, you want to put the name of the object in parentheses, add a dot to access the properties and then enter the property name.

When I typed it in, I got a count property of 1,141. Your number might be different, depending on what versions of Windows and PowerShell you are running. Don't worry too much about these differences now.

Counting, as simple as it is, can come in handy in all sorts of situations. For example, in Exchange installations, you can get a count of how many mailboxes there are by just using:

```
(get-mailbox).count
```

Or maybe you have an Active Directory deployment and you want to know how many users there are in total. Or you want to know how many events there are in an event log. Maybe you want to monitor how many events there are from a certain service or application that writes to the event log — but only when there are errors. A simple monitoring system could use the count property for a given event log, see when it's greater than zero and then do some stuff to alert you, like send an email.

The possibilities are pretty much endless, and many things you will want to do in PowerShell come from looking at the properties of something you're dealing with.

“But Jon,” I hear you asking. “How do I know what sorts of properties an object might have?” After all, when you right-click and select Properties in Windows, the GUI makes it fairly apparent what all of your choices are. Well, keep reading.

Help me!

When I asked folks who were new to PowerShell what they wanted to learn about first, I received a really interesting comment:

I have used PowerShell commands fed to me in their entirety by various postings on the Internet — solutions to particular problems. I can see how powerful it is, but am amazed at how much there must be to remember. Do people remember all of the commands, their syntax, the command options, switches, etc?

This is an excellent question. PowerShell is a big world of cmdlets, but don't despair — each of them is thoroughly documented using the built-in help. You can access it by — wait for it — using a cmdlet called *Get-Help*. For any cmdlet, just precede the cmdlet name with *Get-Help* and you'll get a world of documentation.

Introduction to Windows PowerShell

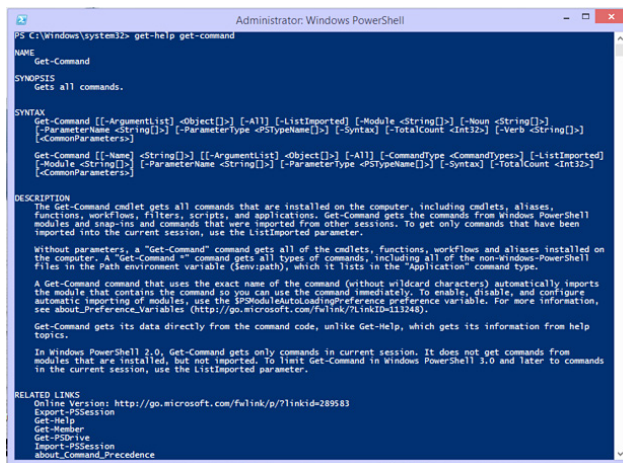
TIP: When you first deploy your system, you'll probably want to run *Update-Help*, which will download the latest PowerShell help files from Microsoft and install them on your system. Be sure to run this as an administrator. It won't work otherwise.

The update process will only take a minute or two, but then you'll be using the latest documentation. You might want to run this every six months or so just to make sure you're not missing anything.

Let's see this in action. We'll go back to our *get-command* cmdlet and ask for some help:

`Get-help get-command`

This is what I see:



```
PS C:\Windows\system32> get-help get-command

NAME
    Get-Command

SYNOPSIS
    Gets all commands.

SYNTAX
    Get-Command [-ArgumentList <Object[]>] [-All] [-ListImported] [-Module <String[]>] [-Name <String[]>]
    [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-Syntax] [-TotalCount <Int32>] [-Verb <String[]>]
    [-CommonParameters]

    Get-Command [-Name <String[]>] [-ArgumentList <Object[]>] [-All] [-CommandType <CommandTypes>] [-ListImported]
    [-Module <String[]>] [-ParameterName <String[]>] [-ParameterType <PSTypeName[]>] [-Syntax] [-TotalCount <Int32>]
    [-CommonParameters]

DESCRIPTION
    The Get-Command cmdlet gets all commands that are installed on the computer, including cmdlets, aliases,
    functions, workflows, filters, scripts, and applications. Get-Command gets the commands from Windows PowerShell
    modules and snap-ins and commands that were imported from other sessions. To get only commands that have been
    imported into the current session, use the ListImported parameter.

    Without parameters, a "Get-Command" command gets all of the cmdlets, functions, workflows and aliases installed on
    the computer. A "Get-Command =" command gets all types of commands, including all of the non-Windows PowerShell
    files in the Path environment variable ($env:path), which it lists in the "Application" command type.

    A Get-Command command that uses the exact name of the command (without wildcard characters) automatically imports
    the module that contains the command so you can use the command immediately, to enable, disable, and configure
    automatic importing of modules, use the $PSModuleAutoLoadingPreference preference variable. For more information,
    see about_Preference_Variables (http://go.microsoft.com/fwlink/?linkid=132248).

    Get-Command gets its data directly from the command code, unlike Get-Help, which gets its information from help
    topics.

    In Windows PowerShell 2.0, Get-Command gets only commands in current session. It does not get commands from
    modules that are installed, but not imported. To limit Get-Command in Windows PowerShell 3.0 and later to commands
    in the current session, use the ListImported parameter.

RELATED LINKS
    Online Version: http://go.microsoft.com/fwlink/p/?linkid=289583
    Get-Help
    Export-PSSession
    Get-Command
    Get-Member
    Get-PSDrive
    Import-PSSession
    about_Command_Precedence
```

PowerShell's Get-help cmdlet.

You can see that the *get-command* cmdlet takes a lot of parameters. If you know DOS at all, you'll remember DOS commands have flags, like *dir /s*, which list the contents of the directory and then all subdirectories, or *move /y* to move things without being required to confirm the move for Every, Single, File. Those DOS flags altered the way a command responded by default.

These cmdlet parameters act in the same way—they affect what the cmdlet does. You can tell *get-command* to give you:

- Only certain types of commands with *-CommandType*
- Commands with certain syntax using *-Syntax*
- Everything using *-all*
- And so on

Most PowerShell cmdlets will accept parameters. You will rarely just use the default output of a command. So it's important to know where to look to find these parameters.

Introduction to Windows PowerShell

What else might be useful to find out about a given cmdlet? If you're sniffing around an area or a topic, the Related Links section can be really helpful; this is where you can find related PowerShell cmdlets that might do something else you want related to the same topic or area.

For instance, if I'm running *get-eventlog* because I want information about the event log, I probably want to do something with that information, so running *get-help get-eventlog* shows me (in the Related Links section) other cmdlets I'm probably going to be interested in learning more about, including:

- Clear-Eventlog
- Get-Winevent
- Limit-Eventlog
- New-Eventlog
- Remove-Eventlog
- Show-Eventlog
- Write-Eventlog

And then I can use *get-help* with each of those suggestions to get more information.

Are you starting to see how you can work your way through a task using the built-in documentation? The *get-help* cmdlet itself has parameters, and one of the most helpful parameters is the *-examples* parameter. This lists out a bunch of example cmdlets that use parameters in various forms and combinations, so you can see the commands in action and build your own customized versions of these cmdlets based on the examples.

Try it yourself. Enter:

```
get-help get-command -examples
```

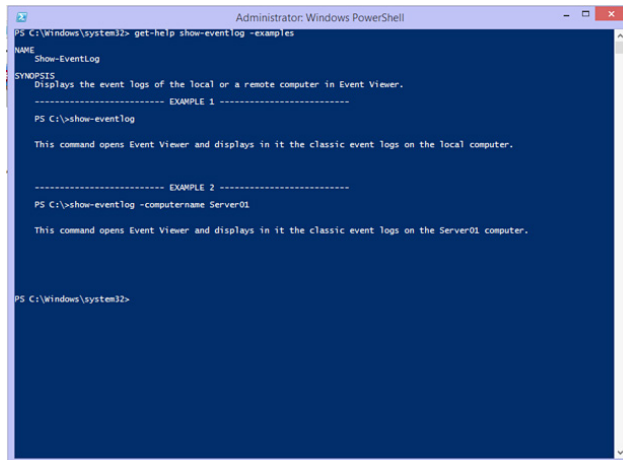
You can see that the examples are well documented and clear. And these are everywhere in the PowerShell universe. Going back to an Exchange environment, here's what I see with *Get-Mailbox*:

```
Machine: SERVER1.corp.2ventures.com
(PS) C:\Windows\system32\get-help get-mailbox -examples
NAME
    Get-Mailbox
SYNOPSIS
    Use the Get-Mailbox cmdlet to view mailbox objects and attributes, populate property pages, or supply mailbox information to other tasks.
EXAMPLE 1
    This example returns a list of all the mailboxes in your organization.
    Get-Mailbox -ResultSize unlimited
EXAMPLE 2
    This example returns a list of all the mailboxes in your organization in the Users OU.
    Get-Mailbox -OrganizationalUnit Users
EXAMPLE 3
    This example returns all the mailboxes that resolve from the ambiguous name resolution search on the string "Chr" if they are in the domain named DCBI. This example returns mailboxes for users such as Chris Hixton, Christian Hess, and Chris Geller.
    Get-Mailbox -Amb Chr -DomainController DCBI
EXAMPLE 4
    This example returns information about the mailbox named Chris, including archive mailbox information.
    Get-Mailbox -Identity Chris -Archive
```

PowerShell's *Get-Mailbox* cmdlet used in an Exchange environment.

And here's what I see for *get-help show-eventlog -examples*:

Introduction to Windows PowerShell

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The prompt is "PS C:\Windows\system32> get-help show-eventlog -examples". The output shows the help for the "Show-EventLog" cmdlet. It includes the name, synopsis, and two examples. Example 1 shows the command "PS C:\>show-eventlog" and its description. Example 2 shows the command "PS C:\>show-eventlog -computername Server01" and its description. The prompt at the bottom is "PS C:\Windows\system32>".

```
Administrator: Windows PowerShell
PS C:\Windows\system32> get-help show-eventlog -examples
NAME
Show-EventLog
SYNOPSIS
Displays the event logs of the local or a remote computer in Event Viewer.
----- EXAMPLE 1 -----
PS C:\>show-eventlog
This command opens Event Viewer and displays in it the classic event logs on the local computer.
----- EXAMPLE 2 -----
PS C:\>show-eventlog -computername Server01
This command opens Event Viewer and displays in it the classic event logs on the Server01 computer.
PS C:\Windows\system32>
```

PowerShell's *Get-Help-Show-Commandlog* cmdlet.

So in answer to the reader's original question — no, everyone doesn't memorize everything all the time.

Now, sure, just like you start to remember familiar DOS commands or even Bash commands in Linux or Unix, you'll remember some things if you use them often enough. But PowerShell's help and the examples that you get right out of the box with no further effort (other than to update the help files as I recommended earlier) will always be there as guides to help you figure out what's going on and how to work your way out of a puzzle.

Something practical

We have covered only the very, very basics here, but let's do something practical before we close up this piece so that you get a real feel for PowerShell and can practice what you read here right away.

Two very common things you do in administering systems are finding processes and killing them. You might currently use Task Manager, or for more detailed work you might use the DOS command *tasklist /svc* to find the process ID for a given service and what it's running, and then use Task Manager to kill it. With PowerShell, we can do the same thing.

Let's say that — purely hypothetically, of course — we have a browser whose name rhymes with Moogle Shrome. It is not well behaved and sucks up a ton of memory if you leave it open overnight. (As mentioned, this is *absolutely* hypothetical.) We need to find it and kill it before it does more damage.

You can probably guess how the cmdlets for working with processes would work. *Get-process* will get a list of processes that are currently running on your system, formatted as a table by default.

PowerShell's *Get-process* cmdlet.

I know from *get-help get-process* that I can specify the name of a process that will display information only about that process:



Introduction to Windows PowerShell

Using a PowerShell cmdlet to get information about a specific process.

And I'm thinking that if *get-process* gets me the information about a process, then using the verb-noun format, *stop-process* would stop one. (I could always check this out using *get-help* or *get-help examples*, but since I'm in a hurry, I proceed without reading the directions and just assume that this cmdlet exists.)

PowerShell's stop-process cmdlet.

My brazenness is rewarded! But now it's asking me for an ID because when I entered the *stop-process* cmdlet, I didn't tell it which process to stop. I could scroll up and look for the ID in the results table from *get-process chrome*. Looks like process ID 2924 is taking up a lot of CPU time, so I'll enter 2924 and hit enter.

If that's the only one I want, I hit Enter again on a blank line. (PowerShell is assuming there might be more than one ID I want to kill and is letting me enter multiple IDs at one time, but to skip out of this anytime you see this, simply enter a blank line.) I immediately note that Googl...ahem, Moogole Shrome has disappeared.

Without the handholding, I could have accomplished the same thing using this sequence:

```
Get-process  
Get-process chrome  
Stop-process 2924
```

And I'd be done.

Wrapping up

Congratulations! We just used PowerShell to kill a process that had gone awry. We puzzled our way through cmdlets, used parameters, remembered ways we could find out related cmdlets and successfully made educated guesses about whether certain cmdlets existed by remembering the verb-noun naming methodology.

Now you have the tools to make a good start at using PowerShell like a pro.

PowerShell for beginners: Scripts and loops

With the advent of Windows Server 10, PowerShell is becoming more important for admins to master, or at least learn. Here's how to get started. By Jonathan Hassell

BACK IN 2008, I wrote a piece called [PowerShell Tips and Tricks](#), which covered the then-relatively new Windows scripting language and some cool things you could do with it. Although PowerShell has been important in the Microsoft ecosystem ever since its release, as [Windows Server 10](#) comes closer to release, we find that many features and deployments are significantly easier and more full-featured when carried out with PowerShell. Simply put, learning the language, or at least being familiar with it, is now a must.

PowerShell is built into Windows, so there is no fee or additional licensing cost. In addition, different server products come with their own PowerShells, too, which expands the universe of things you can do with PowerShell.

I have put together a head-start guide to scripting in hopes that many administrators not yet proficient with PowerShell will use this opportunity to improve their skills and be ready for the next wave of Microsoft software.

An introduction to scripts

The basis of PowerShell is script processes and commands. Once we have the framework of creating our own scripts down, we can add in some of the more advanced logic that involves loops and conditionals.

Scripts in PowerShell are basically just text files with the special filename extension of `ps1`. To create a script, you enter a bunch of PowerShell commands in a sequence in a new Notepad file (or really, you could use any text editor you like), and then save that file as `NAME.ps1`, where `NAME` is a friendly description of your script — with no spaces, of course.

To run a PowerShell script that you already have, you enter in a PowerShell window either:

- The full path (folder and file name) of the script, like so: `c:\powershell\myscrip-`



Introduction to Windows PowerShell

there.ps1

Or

- If your script is in the current directory the console is looking at, use a period and then a backslash, like this: `.\myscripthere.ps1`

Other than that, there is nothing special needed for creating a script in PowerShell. You simply add the commands you like.

Of course, a script probably needs to do more than one thing or you wouldn't need a script for it. Scripts are common in IT already; administrators have been using login scripts for years to get users' desktops and environments configured consistently every time a user logs on. As technology has gotten better, you can script almost anything, from the bare-metal installation of an operating system on a server fresh out of the factory box all the way up to server workloads, including installing Exchange or file server roles.

We obviously won't go that in depth in this piece, but the basic idea behind a script for our purposes is to do two or three things and then end.

To do that, we need to cover a few elements of a script. The first is the element that can change. The second is the element of the script that actually does the dirty work on everything. Let's look at each of the phases.

Making scripts useful, phase 1: Variables

Now if you buy that the whole point of scripting is to do things over and over again in a consistent way, then you have to buy the idea that you want to do the same action to different things. But how do you change the things that are being acted upon? Variables are how. Variables are kind of like holders, and you can put values, words, numbers — basically anything — within them.

In PowerShell, variables always have a dollar sign (\$) before them.

Let me declare — in other words, set up for the first time — a few variables for you right now:

```
$name = 'Jon'  
$number = 12345  
$location = 'Charlotte'  
$listofnumbers = 6,7,8,9  
$my_last_math_grade = 'D+'
```

All I have to do to declare those variables is add a dollar sign, then use whatever name for the variable I want — no spaces are allowed in the name itself — and then a space, an equals sign, another space and then whatever I want the value of the variable to be.

If I want to have a variable with text as its value, I need to add a single quote on either side of the text. (There are some exceptions to this, but again my goal here is to keep it simple so we'll stick with this for now.) You can also just declare a variable without putting a value in it. This kind of “reserves” the name, which is probably more helpful when you're in the middle of developing than it is at any other time.

You know what else you can put in a variable? The output of a cmdlet, which is a

Introduction to Windows PowerShell

cute moniker that refers to the simplest bit of .Net-based code you can execute that actually returns a result, either from the PowerShell prompt or from a script. For example, the **Get-Process** cmdlet lists all processes, while the **Get-PSSnapin** cmdlet shows all current PowerShell snap-ins that enable new functionality.

To find out the total number of cmdlets available on the system, we can use:

```
(get-command).count
```

And at least for the system on which I am writing today's piece, that returned a result of 1,338 cmdlets.

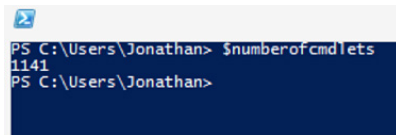
Let's declare a variable to store that count in. We'll call it

```
$numberofcmdlets
```

And let's store in that variable the output of the (get-command).count entry.

```
$numberofcmdlets = (get-command).count
```

PowerShell will tell you the current value of any variable if you just type its name into the prompt, so let's see if that worked:



```
PS C:\Users\Jonathan> $numberofcmdlets
1141
PS C:\Users\Jonathan>
```

Success! Now you can use that variable as part of something else. For a simple example, let's look at the **Write-Host** cmdlet, which simply writes text to the screen of the machine hosting the PowerShell session. **Write-Host** has a bunch of capabilities, but in its simplest form, you can just say:

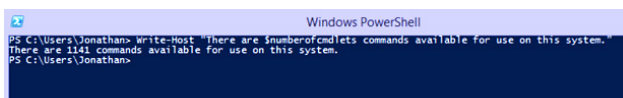
```
Write-Host "Whatever text I want, as long as it is inside double quotes."
```

Indeed, you can cut and paste that line into a PowerShell window and it'll come out exactly like it goes in.

But you can integrate variables with **Write-Host**. You just call them with the dollar sign notation and work them right into your text. For example, I can say:

```
Write-Host "There are $numberofcmdlets commands available for use on this system."
```

And what does PowerShell return to us?



```
Windows PowerShell
PS C:\Users\Jonathan> Write-Host "There are $numberofcmdlets commands available for use on this system."
There are 1141 commands available for use on this system.
PS C:\Users\Jonathan>
```

Let's put variables aside for now, and move on to the next element of scripting: Decision making and looping.

Introduction to Windows PowerShell

Making scripts useful, phase 2: If/Then, Do-While and ForEach

The next phase actually lets you do some magic. We know how to store values in variables now, but we need to do some things to those variables. Let's take a look.

If/Then

The simplest form of decision making in PowerShell is the if/then mechanism; in PowerShell lingo, this is called the “construct.” It basically works like this:

If something is some comparison to something else—> Then do this action.

You format it by putting your comparison in parenthesis, putting a left curly brace alone on a new line, adding the PowerShell cmdlets or actions to perform if that action is true beginning on another new line, and then ending with a right curly brace on a final new line. The key points here are that:

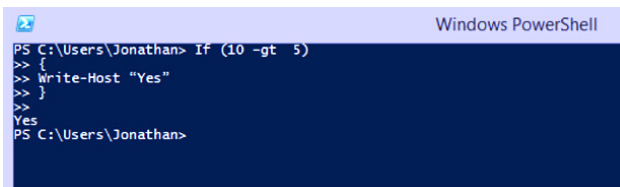
- The comparison statement must have a logical response of either TRUE or FALSE. Think of it as a yes or no question. If you need to do something not based on a yes or no question, then another loop construct is appropriate; we'll cover that in a bit.
- The code that runs if your statement is YES or NO must be within curly braces, and it is best practice to put these curly braces on lines by themselves so that you can match them up when you are writing more complicated scripts.

For example, if I wanted to compare two numbers—let's say 5 and 10—and have PowerShell display whether 10 was greater than 5, then we could write the following if/then construct:

```
If (10 -gt 5)
{
Write-Host "Yes"
}
```

You may already know that `-gt` is the PowerShell switch for “greater than.” We used `Write-Host` in the previous example as well.

If we run this at a PowerShell prompt, we get:



```
Windows PowerShell
PS C:\Users\Jonathan> If (10 -gt 5)
>> {
>> Write-Host "Yes"
>> }
>>
Yes
PS C:\Users\Jonathan>
```

That's kind of simple and probably not going to be a ton of use for you in your administrative duties. To make the construct a little more functional, you can add more “nests” to the If/Then block. Again, these execute in a progression—in programming parlance, this is known as a serial construct, meaning one comparison has to finish, then the next one, and as soon as one finishes, the comparisons stop.

It would look like this:

```
If (10 -gt 11)
```

Introduction to Windows PowerShell

```
{  
  Write-Host "Yes"  
} elseif (11 -gt 10)  
{  
  Write-Host "This time, yes"  
}
```

You should be able to follow that one pretty easily; here is the result.

```
PS C:\Users\Jonathan> If (10 -gt 11)  
>> {  
>> Write-Host "Yes"  
>> } elseif (11 -gt 10)  
>> {  
>> Write-Host "This time, yes"  
>> }  
>>  
This time, yes  
PS C:\Users\Jonathan>
```

The first logical comparison (is 10 greater than 11? No) is false, so PowerShell moves on to the next one via the `elseif` comparison, which is PowerShell parlance for “next, please!” (is 11 greater than 10? Yes), and prints the `Write-Host` line I wrote for that test.

In constructs like these, when you’re testing, it’s best to have different output for each test. If I had chosen `Yes` for both constructs, and then run the script, I would not have been able to tell which comparison test was producing the `Yes` — there’s no differentiation. Here, I added “This time” so I could tell what was actually happening in the script.

You can include a bunch of these `elseif` blocks in your script — theoretically there is no maximum. It’s a great way to establish conditions before you do something. For instance, if I wanted to move mailboxes only where the user’s region was in the United States, then I could use an `IF` statement to get at the mailbox properties and then write the code for the move within the curly braces.

Or maybe I have a machine with a pesky startup problem because of an interaction with an old piece of software, and so I need to write a script that I set off as a scheduled task that checks a service after a minute or two and, if it is stopped (there’s my comparison), starts the service (there’s my code).

Hopefully you can see the applications of this type of PowerShell construct.

Finally, you can choose to include an `else` block in your `if/then` construct, which runs as basically the alternative ending for your script — if all of the `ifs` and `elseifs` do not actually evaluate and run their code, then the `else` block can do something to conclude the script. The `else` block is written at the very end and DOES NOT include a parenthetical comparison statement; you leave it off.

Here’s an example: I might make a series of comparisons like this, and then make a statement of exasperation at the end:

```
If (10 -gt 11)  
{
```

Introduction to Windows PowerShell

```
Write-Host "Yes"  
} elseif (11 -lt 10)  
{  
Write-Host "This time, yes"  
} elseif (20 -gt 40)  
{  
Write-Host "Third time was a charm"  
} else {  
Write-Host "You're really terrible at math, aren't you?"  
}
```

If I run this in PowerShell, this is what I get back from the console:

```
PS C:\Users\Jonathan> If (10 -gt 11)  
>> {  
>> Write-Host "Yes"  
>> } elseif (11 -lt 10)  
>> {  
>> Write-Host "This time, yes"  
>> } elseif (20 -gt 40)  
>> {  
>> Write-Host "Third time was a charm"  
>> } else {  
>> Write-Host "You're really terrible at math, aren't you?"  
>> }  
>>  
You're really terrible at math, aren't you?  
PS C:\Users\Jonathan>
```

That's if/then constructs in a nutshell.

Do While

Do While is the simplest of the looping constructs in PowerShell. A looping construct is basically a piece of code that does the same action over and over to a set of things — in other words, it loops through a set of things, doing something to each of them, until some condition changes or that set of things is exhausted.

There are two main types of looping constructs in PowerShell -- a Do While loop, and another one I will explain in the next section.

Do While is simply a construct that says to PowerShell, "do this to this set of things until some condition I tell you becomes true." It's really as simple as that.

There are two ways to set up this construct. If you want a set of commands to execute at least once, and then as many times as are necessary to satisfy whatever condition you set up, you can simply put Do and a left curly brace on one line, the commands to execute starting on a new line after the left curly brace. And then on a new line, put the right curly brace followed by While and then, within parenthesis, your conditional statement. The conditional statement, again, must be true or false.

For example, let's set up a variable called numbers and give it an initial value of one.

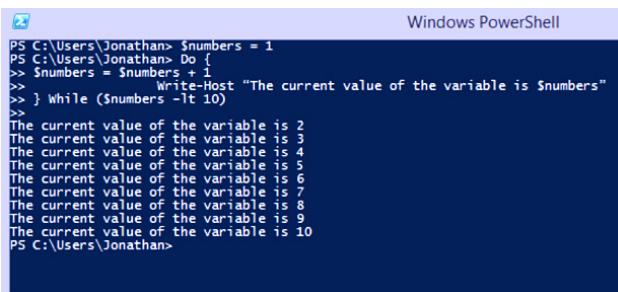
```
$numbers = 1
```

Then, let's set up a simple Do While construct that adds 1 to whatever number is already in that variable, until the variable has the number 10 in it.

Introduction to Windows PowerShell

```
Do {  
    $numbers = $numbers + 1  
    Write-Host "The current value of the variable is $numbers"  
} While ($numbers -lt 10)
```

Here's what that looks like in the console:



```
Windows PowerShell  
PS C:\Users\Jonathan> $numbers = 1  
PS C:\Users\Jonathan> Do {  
>> $numbers = $numbers + 1  
>> Write-Host "The current value of the variable is $numbers"  
>> } While ($numbers -lt 10)  
>>  
The current value of the variable is 2  
The current value of the variable is 3  
The current value of the variable is 4  
The current value of the variable is 5  
The current value of the variable is 6  
The current value of the variable is 7  
The current value of the variable is 8  
The current value of the variable is 9  
The current value of the variable is 10  
PS C:\Users\Jonathan>
```

You can also set up a Do While construct so that your set of commands only executes when the condition is true. You just need to eliminate the do statement, and only use while.

```
While ($numbers -lt 10) {  
    $numbers = $numbers + 1  
    Write-Host "The current value of the variable is $numbers"  
}
```

ForEach

ForEach is the other looping construct. ForEach simply looks at a set of things and pulls out one at a time to look at them and then, if you say so, perform some type of action or set of commands on it.

Here's how to think of this. Let's say you had a list of users sent over from your HR department, a list of employees who had resigned in the previous quarter. You need to disable their Active Directory accounts. You can use a ForEach loop to work on this. You would say: Dear PowerShell...

```
Here's my list of users  
ForEach (user in that list)  
{  
    disable their log on ability in Active Directory  
}
```

Note the familiar curly braces and their placement. (In developer parlance, what I just showed you above in this example is called pseudocode — it's a way to break down how you would write code without taking the time to actually figure out the correct syntax, just as a way of making sure you have a good game plan when you approach a problem for which you need to eventually write code to solve.)

One different part of a ForEach loop is the keyword **in** that lives within that paren-

Introduction to Windows PowerShell

thetical statement. That tells PowerShell to create a single variable to hold the values that come out, one at a time, for your bigger set of things.

Let's look at a real code example. Let's just use a simple set of names in a variable.

```
$names = "Amy","Bob","Candice","Dick","Eunice","Frank"
```

When we make a list within a variable, we have created what is known as an array, which is simply a big sort of matrix thing that PowerShell holds in memory that lets it store a bunch of things at one time.

Let's also initialize a count variable so we get a sense of how the loop is going.

```
$count = 0
```

Then, let's use a `ForEach` loop to count how many names we have. Remember our keyword `in` — we have to create a new variable that we can call anything we want. This holds each single name that comes out of that list of names we have stored in the variable `$names`.

I have called this new variable `$singlename` just to make it clear it is a new variable that just holds a single value that comes out of a list. PowerShell works on that single value and then moves on, grabbing another value from the bigger list, storing it in the new single variable, acting on it based on whatever commands you have put into the loop, and then lathering, rinsing and repeating.

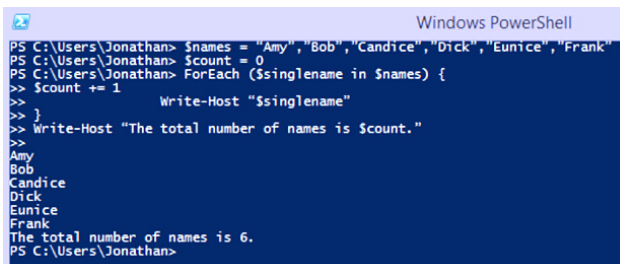
```
ForEach ($singlename in $names) {  
    $count += 1  
    Write-Host "$singlename"  
}
```

The `+=` shorthand basically just says increment the number by whatever interval I put next, in this case 1. I then added a `Write-Host` line with the `$singlename` variable so we can get a glimpse into what value PowerShell is working on in the loop.

Finally, I'll add a simple `Write-Host` line after the end (after the right curly brace, that is) to display the count, so we can actually answer our question.

```
Write-Host "The total number of names is $count."
```

Here's what it looks like all put together and run in PowerShell:



```
Windows PowerShell  
PS C:\Users\Jonathan> $names = "Amy", "Bob", "Candice", "Dick", "Eunice", "Frank"  
PS C:\Users\Jonathan> $count = 0  
PS C:\Users\Jonathan> ForEach ($singlename in $names) {  
>> $count += 1  
>> Write-Host "$singlename"  
>> }  
>> Write-Host "The total number of names is $count."  
>>  
Amy  
Bob  
Candice  
Dick  
Eunice  
Frank  
The total number of names is 6.  
PS C:\Users\Jonathan>
```

That's a `ForEach` loop.



Introduction to *Windows PowerShell*

The last word

That is, in a nutshell, how you can begin creating simple scripts and looping constructs in PowerShell. Use this information to build some scripts of your own today!



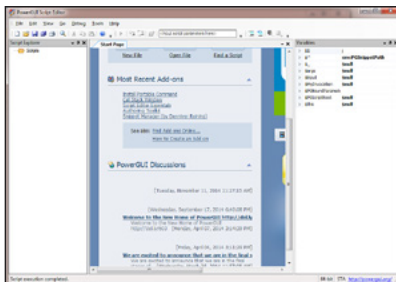
Introduction to Windows PowerShell

9 useful PowerShell tools

Microsoft's mighty Windows administration framework gets even better with the help of these tools and materials. Almost all are free; the one for-fee tool is well worth the cost. By Jonathan Hassell

Why PowerShell?

Ah, PowerShell. A simple blue window and some text has transformed the world of Windows administration from a point-and-click GUI to [scripts that automate everything](#), as well as provide log rotation and identity lifecycle management and which server receives which updates. With everything in the newest versions of Windows Server accessible primarily via PowerShell and only secondarily (and sometimes even not at all) via the server's GUI, PowerShell knowledge has become a must. Sometimes, though, it is difficult to know whether you are proceeding correctly. Luckily, there are other resources available that will help speed you along in your training and your professional responsibilities. In this slideshow, I will highlight 9 resources for immersing yourself in the PowerShell world. Whether you're writing scripts, working in a DevOps-oriented environment or administering software from vendors other than Microsoft using PowerShell, there is something for everyone in this group of resources. And best of all — they are all free, save for one excellent paid product. What are you waiting for? Let's dive in.

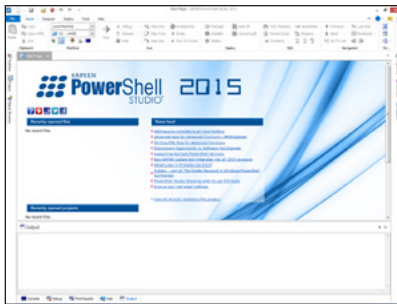


Dell PowerGUI

Presumably left over from Dell's 2012 [acquisition of Quest](#), PowerGUI is a visual complement to PowerShell. It makes assembling scripts and getting things done in PowerShell as simple as selecting cmdlets that are appropriate for your task and then dragging them into place. Perfect for those who are new to PowerShell but have a basic grasp of its concepts, PowerGUI is an easy-to-use script editor that will probably advance your understanding of assembling more complex and advanced scripts quicker than anything else — especially if you are a visual learner. Perhaps the most useful features of PowerGUI are the Power Packs: Pre-built scripts that have been open-sourced by the user community and made available to other PowerGUI users. These range from adding users to managing switches; they can be customized and further improved upon, or simply baked into whatever script you are currently writing, saving the time it would take you to

Introduction to Windows PowerShell

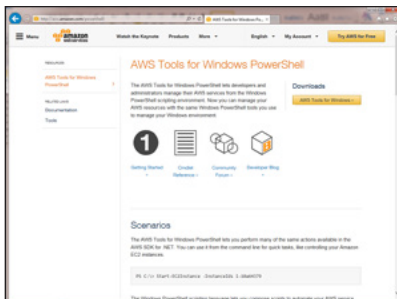
reinvent the wheel. There was once a paid edition of PowerGUI with more advanced features, but that edition was rolled up into the freeware product. PowerGUI does not seem to have been updated for a while, but that does not make it any less useful, and since it is freeware, you have nothing to lose by adding it to your arsenal. [Freeware](#).



Sapien Technologies PowerShell Studio 2015

More advanced PowerShell developers and administrators need more advanced tooling, and PowerShell Studio 2015 from Sapien is the first place to look. When you first open PowerShell Studio, you are immediately reminded of Visual Studio and for good reason: PowerShell Studio is as

much an integrated scripting environment as Visual Studio is an integrated development environment (IDE). Features include: Ribbon, remote debugging support, compiler features that let you turn scripts into executable files, support for multiple versions of PowerShell (useful for targeting scripts to different servers running different levels of the Windows Server operating system), source control for checking in and out script code and support for multiple developers. All of which make this an obvious choice for shops where administrators and developers work together on building advanced PowerShell scripts to handle a variety of scenarios. At \$389 per license, it is a little pricey. But considering all of the product's functionality, if you live in this part of the PowerShell world, it is well worth the cost of admission. [45-day free trial, \\$389 per license](#).



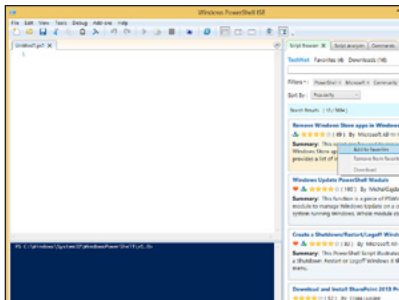
Amazon AWS Tools for Windows PowerShell

It's not just Microsoft that is jumping on the PowerShell bandwagon; even a competitive cloud service like Amazon Web Services recognizes that (a) Windows Server is huge, (b) lots of administrators are learning PowerShell, and (c) anything that lets administrators manage Amazon

services more easily increases the likelihood that an Amazon server will stick in any given enterprise. Thus the AWS Tools for Windows PowerShell were born. With AWS Tools for Windows PowerShell, you can manage virtual machines and service instances that are running in the [Elastic Compute Cloud \(EC2\)](#), or write scripts that automate the management of any workloads you have running in a variety of Amazon services. The tools install a bunch of cmdlets into your Windows PowerShell

Introduction to Windows PowerShell

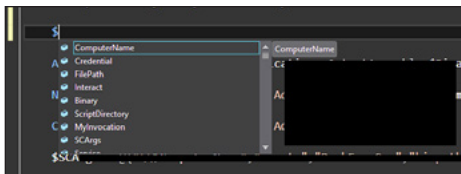
“sphere of influence” and let you manage and script tasks like backing up data from virtual machines in EC2 to the Simple Storage Service (S3) or logging and publishing metrics to the Amazon CloudWatch personal dashboard. If you know PowerShell and you use Amazon cloud services, these tools will be a great addition. [Freeware](#).



Microsoft Script Browser for Windows PowerShell ISE

The problem: You want to do something in PowerShell. You know your outcome. But you do not know how to get there and, further, you have a sneaking suspicion that someone, somewhere out there on the Internet has already figured it out and probably would tell you for free. What

if there were this free magic tool that would scour the TechNet Script Center — probably the most authoritative source for PowerShell scripts on Earth right now — and find scripts that purport to do what you need? That is exactly what Microsoft Script Browser claims to do. It also includes a built-in Script Analyzer function that will read through your scripts and suggest improvements or changes to make based upon scripting best practices. This tool plugs right into the Windows PowerShell Integrated Scripting Environment, which you get for free as part of Windows. You might need to install the feature on Windows client machines, but it should be installed by default as part of the basic Windows Server image. [Freeware](#), requires a Windows license and Windows PowerShell ISE to be installed.



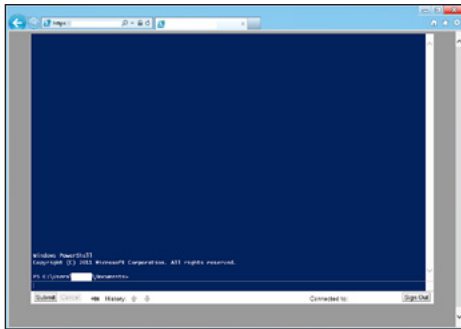
Adam Driscoll's PowerShell Tools for Visual Studio

If you are more on the “dev” side of DevOps, then you probably use Visual Studio as one of your tools

of choice. While Visual Studio has a lot going for it, it does not do a lot with PowerShell out of the box. That is where Adam Driscoll's PowerShell Tools for Visual Studio project comes in. This project integrates within Visual Studio, brings syntax highlighting and colors to the IDE, and adds IntelliSense support for automatically completing syntax elements like variables, cmdlets and arguments as you type within a Visual Studio window. It also extends options for configuring Visual Studio projects so you can keep your scripting efforts organized and together, extends support for scripting arguments with the MS Build compiler and supports script debugging via breakpoint and breakpoint pane support. It also extends some testing features with Pester and PSate test adapters. All in all, this is a free set of resources for making Visual Studio more PowerShell savvy. If you like this after downloading it, consider

Introduction to Windows PowerShell

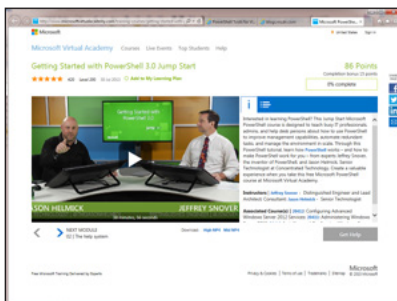
throwing Mr. Driscoll a few bucks for his efforts. [Free](#), with donations solicited. Also [from MSDN](#).



Microsoft Windows PowerShell Web Access, via Control Panel

PowerShell Web Access is like webmail but for PowerShell cmdlets. You log into a webpage that presents a Web-based console where you can run cmdlets, perform operations and do simple remote administration tasks right

over the Internet. There's no need for PowerShell, extensions or cmdlets to be installed on the machine you are browsing with. This means that yes, you can run PowerShell operations from your iPad if you have this feature enabled. Best of all, it is free with a Windows Server license and is built right in. I do not see this in use a lot, but I think it is very handy. As the saying goes, "You might not need this until you need it, but when you do end up needing it, you need it very badly." Be careful, though, as opening this facility up to users outside your network is just an invitation for security problems. Restrict access to the PowerShell Web Access site in IIS to only IP addresses local to your corporate network. Or even better, restrict that access to a few workstations on your local network and perhaps a static VPN address you can use to perform administration tasks remotely. [Free](#). Windows feature, installed through Control Panel / Add and Remove Windows Features. You can also check out Microsoft's [help page on this](#).

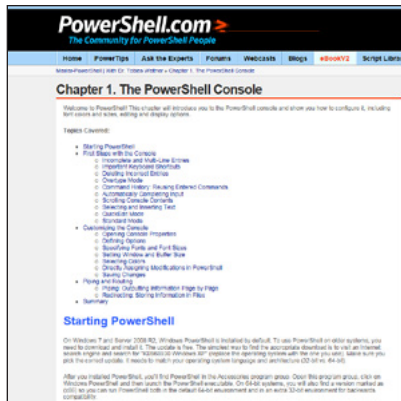


PowerShell Training via the Microsoft Virtual Academy

With great power comes the need for a lot of training. PowerShell is a capable language that can do so much. It marries scripting with development and .NET programming. It comes with a universe of cmdlets. It has its own syntax. And while I have (and will continue to) [explain PowerShell basics training](#) on Computer-

world.com, those pieces just scratch the surface of all there is to know. Fortunately, the Microsoft Virtual Academy contains hours of video training on getting to know PowerShell, using it and making the language work for you. These courses include information from stars such as the father of PowerShell, Jeffrey Snover, and distinguished technologists who have made (new) careers out of understanding every nook and cranny of PowerShell. Perfect for lunch hours. [Freeware](#).

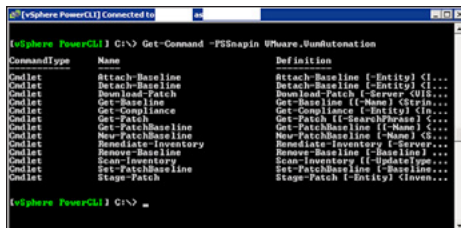
Introduction to Windows PowerShell



Master-PowerShell, an ebook from Dr. Tobias Weltner

If you are a visual learner, then video training is the best way to learn PowerShell. For those of us more language inclined, we can learn from Microsoft MVP Dr. Tobias Weltner in his free ebook cleverly titled *Master-PowerShell*. Weltner covers a lot of ground in his book, including variables, arrays and hashtables, the pipeline, objects, conditions, loops, functions, scripts, error handling, scope, text and regular expressions. Also included:

XML, administrative work using the file system, Registry, processes, services, event logs, WMI and users. He even includes a chapter on .NET and compiling for the developers among us. The book is hosted by Idera, a popular administrative tool developer, and can be found over on the PowerShell.com site, which is a useful community resource in its own right. [Free](#).



VMware vSphere PowerCLI

Like Amazon, VMware has figured out that, in some respects, making nice with your competitors for the benefit of your mutual customers is not a bad thing. To that end, VMware created PowerCLI, a command line-based environment

for managing VMware vSphere resources that integrates PowerShell throughout. The PowerCLI environment is basically a bunch of cmdlets that interact with vSphere and vCloud, and also provides interfaces based on C# and PowerShell for the various APIs that are exposed by the VMware products. If you are a VMware shop and want to get your hands on PowerCLI, head over to [this link](#). Is it not great when everyone plays nicely together in the sandbox? [Freeware](#), with a [free cmdlet reference available](#).

Understanding and using objects in PowerShell

How to use PowerShell objects, how to tease more info and functionality out of them and how objects can be useful in scripting scenarios. By Jonathan Hassell

ONE OF the things most people do not realize about PowerShell, at least up front, is that PowerShell is based on the .NET Framework, which means that PowerShell can be considered a programming language. In fact, each response you get from running a cmdlet in PowerShell, no matter how simple or complex that cmdlet may be, is actually a .NET object. It might look like text to you, but it can be programmatically manipulated in ways that Linux and UNIX command line diehards can only dream about.

In this piece I'll focus on using PowerShell objects, how to tease more info and functionality out of them, and how objects can be useful in scripting scenarios.

What is an object?

It would probably help to know what an object is so that you can understand just how useful this capability of PowerShell is.

Objects are essentially known quantities of something that programming languages can use, interact with, perform computations and transformations on, and in general “consume.” Technically, an object is simply the programmatic representation of anything. Objects are usually considered as two types of things: *Properties*, which simply describe attributes of whatever the .NET object is representing, and *methods*, which describe the types of actions (think verbs, or short instructions) that the .NET object can undertake.

For example, let us consider a car as an example. If we were making a car into a .NET object, then its properties would include its engine, doors, accelerator and brake pedals, steering wheel and headlights. Its methods would include turn engine on, turn engine off, open doors, close doors, press accelerator, release accelerator, turn steering wheel left, turn steering wheel right, turn on headlights, turn off headlights, turn on brights and turn off brights. (That is not an exhaustive list, but it should serve to demonstrate to you that the properties of the car are a description of its components, and the methods of the car describe how you can operate and interact with the properties.)

In PowerShell, it is a simple matter to see an object's properties and methods: Just use the `Get-Member` cmdlet to view them. You can do this by piping the output of a

Introduction to Windows PowerShell

cmdlet. Remember that output is an object to the Get-Member cmdlet, like this:

GET-COMMAND GET-MEMBER		
TypeName: System.Management.Automation.AliasInfo		
NAME	MEMBERTYPE	DEFINITION
Equals	Method	bool Equals(System.Object obj)
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
ResolveParameter	Method	System.Management.Automation. ParameterMetadata ResolveParameter(string name)
ToString	Method	string ToString()
CommandType	Property	System.Management.Automation.CommandType CommandType {get;}
Definition	Property	string Definition {get;}
Description	Property	string Description {get;set;}
Module	Property	psmoduleinfo Module {get;}
ModuleName	Property	string ModuleName {get;}
Name	Property	string Name {get;}
Options	Property	System.Management.Automation. ScopedItemOptions Options

You can see in the middle column that the different methods and properties are delineated, but what is that third column? Those are called data types, and they basically show the classification of answer that will be returned by that method or property (for instance, telling if something is yes or no or true or false would be a Boolean type, whereas a response consisting of text would generally be a string). We will see data types coming into action a little later in our [PowerShell series](#), so stay tuned for that.

You will find as you get into more day-to-day administration with PowerShell that you will be using this Get-Method cmdlet a lot, and the reason is because it is going to tell you exactly how you can interact with various objects.

For example, let us talk about finding files on a shared drive of a certain type. How do you end up knowing exactly what cmdlets and syntax to use to work out how to find specific files with a certain type of file extension? It's through the use of these methods and properties and the PowerShell pipeline, which of course pipes objects and responses through from one cmdlet to the next.



Introduction to Windows PowerShell

An example

Say you have been [infected with Cryptolocker](#) on one of your business' machines. This is a nasty bug that is ransomware; it is malware that silently encrypts the files it finds in a couple of places on your machine (My Documents and mapped drives being a couple of them). And then the bug makes you pay several hundred dollars in untraceable Bitcoin or Green Dot prepaid debit cards to get the key to decrypt them. You either pay up or you lose access to your files.

In our example, let's assume you were able to find the infection before it had the time to encrypt all your files. You immediately shut down the machine, so the encryption process stopped, but as part of your diagnosis of what happened, you need to figure out a list of all the files that were modified in the last day or so. There is a cmdlet called `Get-ChildItem`, which is your tool of choice when you want to grab something out of a giant container of items -- in this case the file system.

So we know to start with `Get-ChildItem`, but how do we know what parameters to put along with it?

First, we can check out `get-help get-childitem`, which will show us that the syntax starts off with `-Path`, so we know that if we are concerned with potentially encrypted data at the mapped drive S: where shared documents are stored, we would use `-Path S:\` to establish where to look.

But what about subdirectories, subfolders, and any sort of nested structure we want to examine as well? From `get-help get-childitem` we also see the `-Recurse` parameter; recursive checking means the program will start at the top and then "recurse," or walk down, the hierarchy of files until everything has been properly examined. We'll add that to the cmdlet as well.

That brings us to this partial cmdlet:

```
Get-ChildItem -Path S:\ -Recurse
```

You can actually run that, and PowerShell will spit out a list of every single file on the S: volume separated out by subdirectory. But we need to examine more about that huge list of files, so we will use the pipeline function to send that output into another cmdlet.

But what cmdlet helps us select a portion of a big set of data for further processing? That is the job of the `Where-Object` cmdlet.

So our cmdlet takes on further shape and body:

```
Get-ChildItem -Path S:\ -Recurse | Where-Object
```

Remember that we add in curly braces, and then within them we can use the `$_`, or as I like to affectionately call it, "that thing," to represent the output of a previous cmdlet that is being piped into a new cmdlet. Then, we add a period or dot and then the name of a property of that object that is represented by `$.`

Here is what we have so far:

```
Get-ChildItem -Path S:\ -Recurse | Where-Object {$_.
```

But what is `Where-Object` going to filter? That's where we need to find out what

Introduction to Windows PowerShell

the properties of Get-ChildItem are; we can use those properties to “tune the antenna,” so to speak, of Where-Object so that it is filtering on the right criteria. To find those properties, let us consult with Get-Member.

GET-CHILDITEM GET-MEMBER		
TypeName: System.IO.DirectoryInfo		
NAME	MEMBERTYPE	DEFINITION
LastAccessTime	Property	datetime LastAccessTime {get;set;}
LastAccessTimeUtc	Property	datetime LastAccessTimeUtc {get;set;}
LastWriteTime	Property	datetime LastWriteTime {get;set;}
LastWriteTimeUtc	Property	datetime LastWriteTimeUtc {get;set;}
Name	Property	string Name {get;}
Parent	Property	System.IO.DirectoryInfo Parent {get;}
Root	Property	System.IO.DirectoryInfo Root {get;}
BaseName	ScriptProperty	System.Object BaseName {get=\$this.Name;}
TypeName: System.IO.FileInfo		
NAME	MEMBERTYPE	DEFINITION
IsReadOnly	Property	bool IsReadOnly {get;set;}
LastAccessTime	Property	datetime LastAccessTime {get;set;}
LastAccessTimeUtc	Property	datetime LastAccessTimeUtc {get;set;}
LastWriteTime	Property	datetime LastWriteTime {get;set;}
LastWriteTimeUtc	Property	datetime LastWriteTimeUtc {get;set;}
Length	Property	long Length {get;}
Name	Property	string Name {get;}
BaseName	ScriptProperty	System.Object BaseName {get-if (\$this.Extension.Length -gt 0){\$this.Name.Re...
VersionInfo	ScriptProperty	System.Object VersionInfo {get=[System.Diagnostics.FileVersionInfo]::GetVer...

Note we have two tables of information returned: One for type System.IO.DirectoryInfo, and the other for System.IO.FileInfo. Since we are looking for information on specific files, we will use the latter.

Looking at that second table, we see two properties that might be interesting to us for completing our task: LastWriteTime and LastWriteTimeUtc. This is what we're



Introduction to Windows PowerShell

looking for! We need the last time that a file was written to.

In this case, just to make things simple, we will use `LastWriteTime` rather than worrying about converting timezones to Greenwich Median Time, although you might have a specific purpose for doing so as you progress in your scripting capabilities.

So to put together our fuller picture, here is where we are:

```
Get-ChildItem -Path S:\ -Recurse | Where-Object {$_.LastWriteTime
```

So we have identified the last write time, but we obviously need to do something with that; we need to ask ourselves, in constructing this command, the question: “Where the last write time is *what*, exactly?” So we need a comparison operator.

You may recall from a [previous PowerShell story](#) that we can use `-lt` for “less than” and `-gt` for “greater than.” So in order to figure out what was written in the last day or so, we can pick a date two days ago. In this example, today is May 14, 2015, so if I’m trying to figure out what files have been touched in the last 24 hours, I would want to know files where the last write time is greater than May 12, 2015.

We write this out in standard MM/DD/YYYY format and then enclose it in quotes as it is considered a string. Then we will add the closing curly brace because our comparative clause is complete, and we have the following cmdlet built:

```
Get-ChildItem -Path S:\ -Recurse | Where-Object {$_.LastWriteTime -gt "05/12/2015"}
```

Run that, and you will get a list of every file on the S: volume that has been written to on 5/12/2015 or after -- exactly what we were looking for. And we did that by understanding that (a) the output of `Get-ChildItem` is an object, and (b) we can find the properties of the `Get-ChildItem` output object using `Get-Member` and use those properties to (c) pipe to `Where-Object` to find specific information about a subset of that output.

Extrapolating how to use objects

There are all sorts of convenient ways to use objects and their properties and methods. With all output being an object, it means you can address all sorts of attributes and characteristics of whatever it is you are working on.

For instance, you can display information in a table format that eliminates all of the other facts you do not have any interest in and laser focuses on the facts in which you are interested. For example, let’s look at what is available for `Get-Service`.

```
Get-Service | Get-Member
```

If I run that, I will see in the table that results that `Status` is a property and `Start` and `Stop` are methods. So if I wanted to find out all the services on a machine that were in the `Stopped` state, and then start those services, I might want to build the following cmdlet:

```
Get-Service | Where-Object {$_.Status -eq "Stopped"} | Start-Process.
```

What if I wanted to find all of the Exchange mailboxes that been created in my lab



Introduction to Windows PowerShell

Exchange environment and then delete those mailboxes because I am done with my experiment and want to restore my test deployment? First, I would want to see the properties available for the **Get-Mailbox** cmdlet, a core cmdlet of Exchange or Office 365:

```
Get-Mailbox | Get-Member
```

I would see, among dozens of other properties, the **WhenChanged** property. This might work, so I would test this out:

```
Get-Mailbox | Format-List name,WhenChanged
```

This gives me a list of mailboxes with the mailbox-friendly name and the value of the **WhenChanged** property. Looks like what I need, so I will modify the above cmdlet not to display a list but to receive the output of **Get-Mailbox** into a **Where-Object** filter, where I will grab the **WhenChanged** output and pass only those that meet my comparison criteria via the pipeline to the **Remove-Mailbox** cmdlet for deletion. It ends up looking like this:

```
Get-Mailbox | Where-Object {$_.WhenChanged -gt "05/07/2015"} | Remove-Mailbox
```

Voila.

The last word

Objects are powerful differentiators that make PowerShell a rich and capable command-line environment. Understanding how to use objects and dig into their properties and methods unlocks the entire universe of PowerShell's abilities for you. Take some time to play around with this.

Scriptshow: Performing five common administrative tasks using Windows PowerShell

Want to add a bunch of users without going out of your mind? We show you how to do that, and more. By Jonathan Hassell

SOMETIMES IT'S easiest to learn something new simply by using it, and to my mind PowerShell is no exception. Often we discover new capabilities and features in looking at what tasks other people are accomplishing using PowerShell, and specifically, looking at how they're using the scripting language.

In this scriptshow, I take five common tasks and show how to accomplish them using [PowerShell](#). The tasks are:

- [Adding a user](#)
- [Deleting a specific attachment](#) (like one that contained in a virus or malware payload) from a set of Exchange mailboxes
- Handling the [mailing-list deletion](#) of employees who are leaving the company for any reason
- [Working with CSV files](#) within PowerShell
- [Connecting to certain Microsoft cloud services](#) from your on-premises servers

I provide the cmdlets or a script, and then walk you through how I put the cmdlets or scripts together so that you can see the logic of why the scripts work the way they do. You can use these as a launchpad of sorts for further customization or for creating your own daily administrative task scripts, whatever you would find useful. I hope this gives you a real taste of the practical applicability that the [PowerShell scripting language](#) can bring to your IT life.

With that said, let us get on with it!



Introduction to Windows PowerShell

1. Adding users

Have you ever had a batch of users you needed to create accounts for, but you did not want to page through the wizards in Active Directory Users and Computers? This kind of rote, repetitive task is exactly what Windows PowerShell is designed to handle.

```
Import-Module ActiveDirectory
Import-Csv "C:\powershell\users.csv" | ForEach-Object {
    $userPrincipal = $_.samAccountName + "@yourdomain.local"
    New-ADUser -Name $_.Name
    -Path $_.ParentOU"
    -SamAccountName $_.samAccountName"
    -UserPrincipalName $userPrincipal
    -AccountPassword (ConvertTo-SecureString "cheeseburgers4all"
    -AsPlainText -Force)
    -ChangePasswordAtLogon $true
    -Enabled $true
    Add-ADGroupMember "Office Users"
    $_.samAccountName";
}
```

In this script, we use the Import-CSV cmdlet, which knows how to read .CSV-formatted files. We tell the Import-CSV cmdlet that each row of the CSV data located in C:\powershell called users.csv contains information in three columns: The Name of the user; the samAccountName of the user, which is basically the login ID for the user; and the organizational unit (OU) of Active Directory that the user needs to live in.

We're also telling the cmdlet that we are using the column samAccount Name to create the login ID for the user by marrying the value that lives in that column with the string "@yourdomain.local" to complete the user principal name (UPN).

From there, we loop through the file using ForEach-Object and send that assembled string (which is stored in the PowerShell variable called \$userPrincipal). We assign the default password to each user as "cheeseburgers4all" and then set the Active Directory flag to require the user to change the password at first logon. At the end of the script, we then add all of these accounts to the Active Directory security group called Office Users.

2. Deleting dangerous or objectionable content from Exchange mailboxes

I was inspired by PowerShell MVP [Mike Robbins' post](#) on removing phishing messages from Exchange mailboxes. In this day and age I think [Cryptolocker](#) and CryptoWall ransomware infections are much more nefarious than phishing. The most recent infections go after network drives and are not well picked up and covered by client anti-malware solutions, so if you are not careful you could well pick up an infection.

For this reason, when you see a suspect message, you might want to just get it out



Introduction to Windows PowerShell

of any mailbox that it is located in — a kind of mass deletion, if you will. If you are running Exchange 2010 or later, you can take care of that from within a PowerShell window.

```
Add-PSSnapin -Name  
Microsoft.Exchange.Management.PowerShell.E2010  
Get-Mailbox -ResultSize Unlimited |  
Search-Mailbox -SearchQuery 'Subject:"*Please review the attached invoice*"' -Delete-  
Content |  
Where-Object {$_.ResultItemsCount}
```

In this script, we add the Exchange tools to our PowerShell window and then put two cmdlets together. The first one is a generic Get-Mailbox cmdlet and we also let PowerShell know that we are targeting all of the mailboxes on the system, so we tell it to give us an unlimited result size.

The second cmdlet searches the content within the mailbox and searches the subject field of every message inside each mailbox for the string we provide in the cmdlet parameter. In this case, “Please review the attached invoice” is actually the subject line of a Cryptolocker infection message I just received as I was writing this. The -DeleteContent eliminates the message, and the Where-Object controls the display of the results within the console window.

Before you do this, you might consider adding the -whatif flag to this transaction so that you can see the impact of the cmdlet’s intended deletion across your entire deployment. Also consider the performance implications: PowerShell searching this way is not, as we would say in the South, too terribly efficient, so for a large organization with tens of thousands of mailboxes, you can expect this operation to consume a fair amount of resources for a while.

3. Elegantly handling departed employees and their distribution list memberships

It happens in every organization: Employees leave. They are terminated, they leave voluntarily, they get another job, they retire. Whatever the reason, you need to deal with their accounts. If your organization is like many others, users wind up embedded in tons of distribution lists per department, per project, per location and so on.

We often find departed employee accounts still around, just without any rights or security group memberships. Most identity-lifecycle best practices suggest you should not simply delete accounts when employees leave; often, their mailboxes live on as shared resources for the remaining employees who might need to unlock some data stored within them.

However, these mailboxes can quickly fill up with distribution list messages that are completely unnecessary. So how do you keep a mailbox active but find all of its various distribution list memberships and unsubscribe from them? That’s where this set of cmdlets comes in.

```
New-DistributionGroup -Name "Sayonara" -OrganizationalUnit "yourdomain.local"
```



Introduction to Windows PowerShell

```
-SamAccountName "Sayonara" -Type "Security" Import-CSV separatedemployees.csv |  
ForEach {Add-DistributionGroupMember -Identity "Sayonara" -Member $_.Name}  
$groupstounssubscribe=get-distributiongroup -filter {DisplayName -ne "Sayonara"}  
Get-DistributionGroupMember Sayonara | remove-distributiongroupmember $group-  
stounssubscribe
```

First, we create a new distribution group called Sayonara, the members of which will be the accounts of departed employees. We will then procure a CSV file from human resources that lists their user principal names. We will feed that file into PowerShell, again using the Import-CSV cmdlet, and then say that for every entry (row) in that CSV file, we should add that login ID to the distribution group called Sayonara.

After this, we initialize a variable called groupstounssubscribe. To populate this variable we ask PowerShell to get a list of all Exchange distribution groups, and then filter it down to only those in which the name is not equal to Sayonara. In other words, the lists stored in this variable will be all lists except our new Sayonara list.

In the final step of this set of cmdlets, we ask PowerShell to grab all of the names within the distribution group Sayonara — these are the ones we want to remove from the other groups — and then pipe that list into the remove-distributiongroupmember cmdlet using the list of groups (except Sayonara) to compare against.

What have we accomplished? All the accounts that are a member of Sayonara will get removed from any distribution group that is NOT Sayonara. So the only new mail a departed employee account's mailbox will receive is mail addressed directly to that mailbox. A neat and tidy solution.

(Hat tip to [this post by David Shackelford](#) for the inspiration.)

4. Create a new comma separated values (.CSV) file and populate it with data

This script is fairly simple but it has a number of interesting implications and is very easy to modify for your specific scenarios. We have used the Import-CSV cmdlet a couple of times in this scriptshow already, but I want to show that PowerShell can also write to CSV files as well, which is really useful to get data out of a system, play around with it in Excel and then re-import it into another cmdlet later.

```
Get-Mailbox | Select-Object  
Name,OrganizationalUnit,WindowsEmailAddress | Export-CSV  
C:\powershell\export.csv
```

In this case, what we are doing is using the Exchange Get-Mailbox cmdlet to get a list of all mailboxes on a deployment. We will pipe that output to the Select-Object cmdlet, which grabs specific parts of whatever it is sent; in this case we are getting the name, organizational unit and default email address properties of each mailbox. And then we are piping just those properties over to the Export-CSV cmdlet, which will write them conveniently to the CSV file at the directory path I included above.

If you are wondering how you can easily grab all of the properties you can use



Introduction to Windows PowerShell

within a CSV, just use a get cmdlet and format the output as a list. For example, “get-mailbox jhassell | fl” will show you all of the different properties you can use with the Select-Object cmdlet in the example above to populate the columns in your CSV file.

5. Easily connect to Exchange Online or Office 365 from your hybrid deployment

If you are running a hybrid Exchange deployment, chances are you are connecting up to the Office 365 portal a lot. If you’ve tried to do a lot of administrative work with PowerShell in this scenario, you know it is a bit of a rigmarole to set up the remoting necessary to run PowerShell cmdlets against the Office 365 servers. Below, I’ve created a script that takes care of the setup for you, so that when you are ready to go, you just run the script and enter your Office 365 administrative credentials.

```
$URL = "https://ps.outlook.com/powershell"  
$Credentials = Get-Credential -Message "Enter your Exchange Online or Office 365 administrator credentials"  
$CloudSession = New-PSSession -ConfigurationName Microsoft.Exchange -ConnectionUri $URL -Credential $Credentials -Authentication Basic -AllowRedirection -Name "Office 365/Exchange Online"  
Import-PSSession $CloudSession -Prefix "365"
```

First off, we declare a variable to store the location on the Internet where we’re sending all of these cmdlets — think of it like a Web service. Then, we set up a variable to securely hold our username and password. The Get-Credential cmdlet pops up a window where you can enter credentials, and the variable will hold those credentials as secure strings. The third variable starts a new PowerShell remoting session using the specific remoting language necessary to connect up to Office 365 or Exchange Online (this works for both offerings). Finally, the Import-PSSession merges that session with your current console, letting you work directly within it.

This particular script is specific to hybrid deployments because sometimes namespaces for cmdlets collide. PowerShell does not always know immediately how to sort out — say, if you ran New-Mailbox — whether you wanted to create that new mailbox on your local deployment or up in the cloud.

To fix this, this script loads the Office 365 namespace of cmdlets with the prefix 365. So all Exchange cmdlets that should run in the cloud should use the 365 prefix, a la New-365Mailbox or Get-365DistributionGroup. All Exchange cmdlets that should run on your local deployment should be left as they are by default. This makes it very easy to distinguish one from the other.

If you want to run this script in a purely cloud environment, however, you can just remove the prefix “365” from the last line of the script and everything will return to its default.

Remember, to save this as a script, just put the cmdlets above into a text file and then save the file with an extension of .PS1. Then, from the PowerShell console window, type in .\script.ps1 (that’s period, backslash, name of file) to run the script.